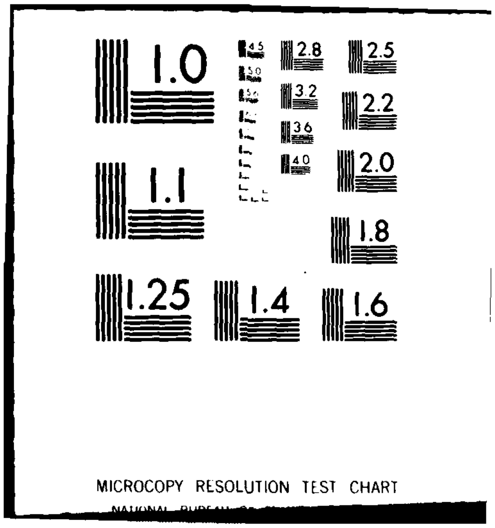
AD-A092 230 OHIO STATE UNIV COLUMBUS COMPUTER AND INFORMATION SC--ETC F/G 9/2
DESIGN AND ANALYSIS OF RELATIONAL JOIN OPERATIONS OF A DATABASE--ETC(U)
SEP 80 D K HSIAO, M J MENON N00014-75-C-0573
UNCLASSIFIED OSU-CISRC-TR-80-8 NL

1 of 1
AD A
092230



MICROCOPY RESOLUTION TEST CHART

NATIONAL BUREAU

TECHNICAL REPORT SERIES

12

COMPUTER & INFORMATION SCIENCE RESEARCH CENTER

THE OHIO STATE UNIVERSITY COLUMBUS, OHIO

(OSU-CISRC-TR-80-8)

12

DESIGN AND ANALYSIS OF RELATIONAL JOIN OPERATIONS OF A DATABASE COMPUTER (DBC)

by

David K. Hsiao
and
M. Jaishankar Menon

Work performed under

Contract N00014-75-C-0573

Office of Naval Research

Computer and Information Science Research Center

The Ohio State University

Columbus, OH 43210

September 80

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER (14) OSU-CISRC-TR-80-8

2. GOVT ACCESSION NO. AD-A092 230

3. RECIPIENT'S CATALOG NUMBER

4. TITLE (and Subtitle) (6) *Design and Analysis of Relational Join Operations ~~in~~ of a Database Computer (DBC)*

5. TYPE OF REPORT & PERIOD COVERED (9) Technical Report

6. PERFORMING ORG. REPORT NUMBER

7. AUTHOR(s) (10) David K. Hsiao
M. Jaishankar Menon

8. CONTRACT OR GRANT NUMBER(s) (15) N00014-75-C-0573

9. PERFORMING ORGANIZATION NAME AND ADDRESS

Office of Naval Research
Information Systems Program
Arlington, Virginia 22217

10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS 4115-A1

11. CONTROLLING OFFICE NAME AND ADDRESS (11) Sep 80

12. REPORT DATE

13. NUMBER OF PAGES 89

14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office) (12) 96

15. SECURITY CLASS. (of this report)

15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Scientific Officer	DDC New York Area
ONR BRO	ONR 437
ACO	ONR, Boston
NRL 2627	ONR, Chicago
ONR 1021P	ONR, Pasadena

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Database computer, post processing, equality join, m-way join, inequality join, A-memories, B-memories, C-memories, associative memories, queueing analysis.

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

This paper repeats in detail the algorithm [4] earlier employed in the database computer (DBC) in order to perform relational equality joins. Two improvements of the algorithm are proposed herein. The first improvement is achieved by adding a new memory component, called the C-memory, for each processor of the algorithm. The C-memory is used to store the results of the join operation. Also, it is a vital component during the execution of a m-way join operation. The second improvement involves the replacement of a single associative memory (AM) by several smaller associative memories (ams). Two advantages accrue as a result of this re-

407586

placement. First, we show that each of the ams needs to operate at a slower rate than the single AM. This implies that we may use random-access memory (RAM) to realize ams, instead of the true and expensive associative hardware. The second advantage is due to an improvement in the complexity of the join operation when the ams are used. It is shown that the time complexity of the join is linear in the cardinalities of the source, target and result sets (relations). We therefore postulate that no join algorithm can have better time complexity than this.

The paper also provides a thorough queueing analysis of the join algorithm as it is carried out in a specialized DBC component known as the post processor (PP). In conducting the queueing analysis, we ignore the presence of the associative memories (i.e., ams), since their presence merely speeds up the algorithm. We also intend to consider a post processor (PP) without the use of the associative memories, because we felt that the use of associative memories may make the post processor rather expensive. In either case, the throughput achievable by the PP without the use of associative memories can be used as the lower bound of the throughput achievable by the join operation.

PP consists of a number of interconnected processors, each of which is associated with three sets of memories. The queueing analysis shows that if PP of DBC is to perform joins at a rate commensurate with the output rate of records (i.e., relational tuples) from the mass memory (MM) of DBC, then the memories associated with PP must satisfy certain constraints with respect to speed and size. In this respect, some constraints are more crucial than others. The paper clearly indicates the order of importance of the various constraints.

The paper also indicates how to select an optimum chip for use in PP, where the design is considered optimal if it provides the fastest speed for a certain fixed cost. The method is one that must be employed by DBC systems designers in order to arrive at the optimal values for

(1) the number of processors, and
(2) the sizes of the memories to be integrated on a single chip.

Such a chip is called an optimal chip.

Finally, the paper describes extensions of the DBC method. The extensions enable PP to perform relational inequality joins and m-way joins. We conclude with a comparison of the DBC join method, with the join schemes proposed on some other database machines.

PREFACE

This work was supported by Contract N00014-75-C-0573 from the Office of Naval Research to Dr. David K. Hsiao, Professor of Computer and Information Science, and conducted at the Computer and Information Science Research Center of The Ohio State University. The Computer and Information Science Research Center of The Ohio State University is an interdisciplinary research organization which consists of the staff, graduate students, and faculty of many University departments and laboratories. This report is based on research accomplished in cooperation with the Department of Computer and Information Science. The research contract was administered and monitored by The Ohio State University Research Foundation.

TABLF OF CONTENTS

1. INTRODUCTION

The database computer (DBC) is intended to support a number of well-known data models [14, 15, 16]. In order to support the relational model [1] of data, the join operation has been provided for in DBC [4]. This paper will first repeat how the relational equality join is performed in DBC. We will then propose two improvements of the method originally proposed [4]. Analysis of the improved join operation is made, indicating that the time complexity of the operation is linear in the input and result sets(relations). Finally, we suggest methods for doing inequality joins and m-way joins by extending the improved join operation.

We will begin by briefly describing the architecture of DBC. This is followed by a description of how the equality join is performed in a specialized unit of DBC known as the post processor (PP). The PP consists of a number of interconnected processors, each of which is associated with three sets of memories. The interconnection scheme is simpler than the one proposed in [4] and is due to our recent study in [5]. The number of memories attached to a processor is also different from the number originally proposed in [4]. The modification is brought about by the addition of a new memory per processor called the C-memory. This memory is used to store the results of the join operation. It is also a vital component in the execution of the m-way join operation to be defined and described in Section 6.

Next, we describe a modification to the method of doing equality joins. This modification involves the replacement of a single associative memory (AM) by several smaller associative memories (ams). Two advantages accrue as a result of this modification. First, we show that each of the ams is required to operate at a slower rate than the one single AM. This implies that we may use random-access memory (RAM) to realize ams. The second advantage is an improvement in the time complexity of the join operation which may turn out to be as fast as any join algorithm can be.

We then provide a thorough queueing analysis of the join operation. By arriving at closed-form equations for various parameters such as the sizes of the memories associated with the processors, we can analyze the memory sizes for various memory speeds. We can also present a design algorithm which allows the DBC systems designers to obtain an optimal chip for use in PP. A chip is optimal if its use in PP provides the best speed of join operation for a certain fixed cost.

A section on the extension of the equality join algorithm for doing inequality joins and m-way joins is included. And finally, a section is provided for comparing the DBC join methods with various other methods proposed in the literature.

2. A BRIEF REVIEW OF DBC ARCHITECTURE

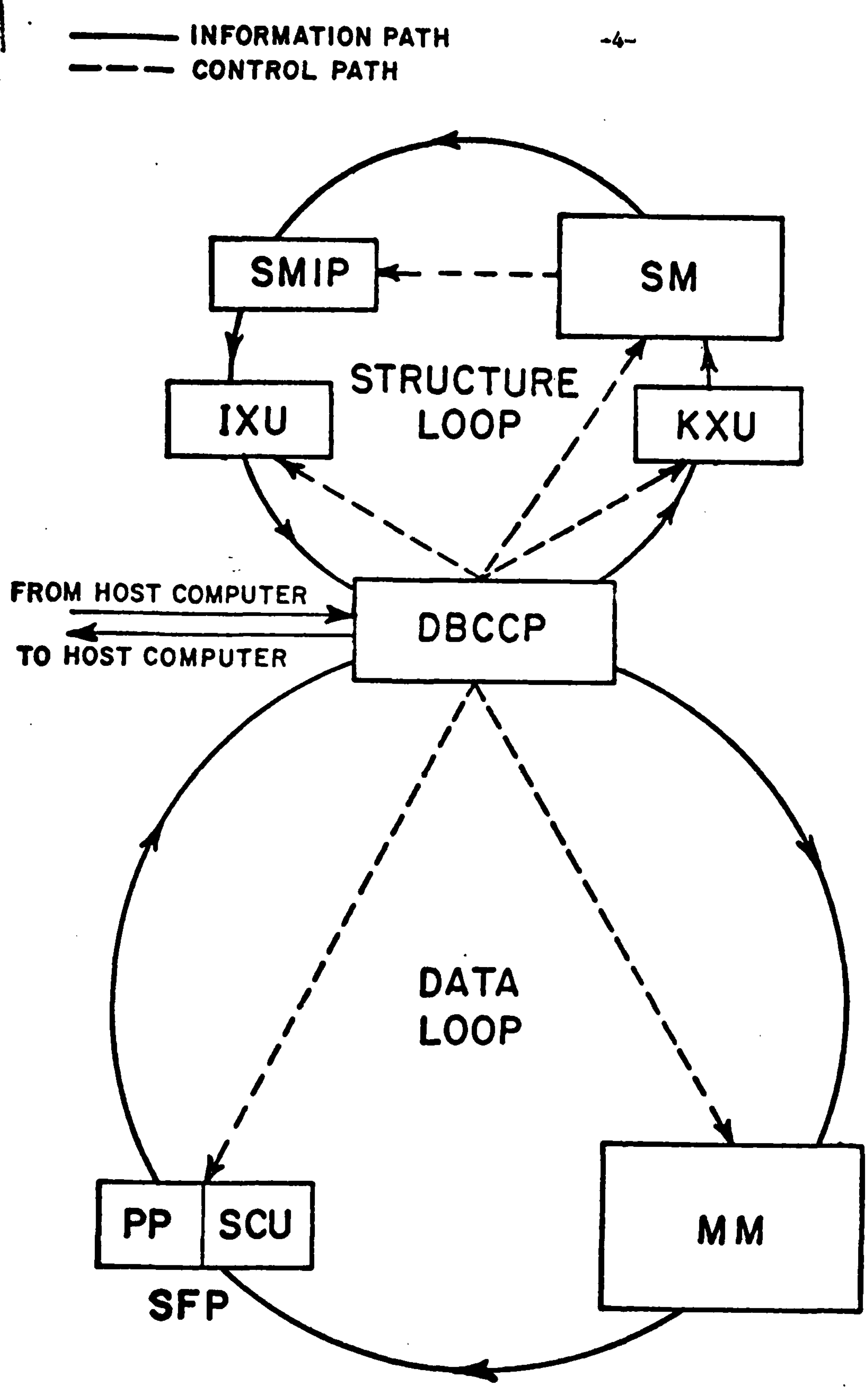
Figure 1 shows the schematic architecture of DBC. It consists of two loops of memories and processors, namely the structure loop and the data loop. The data loop is composed of two components: the mass memory (MM), and the security filter processor (SFP). MM is the repository of the database and is made of modified moving-head disks where all the tracks of a cylinder may be read in parallel in a single disk revolution. This modification is termed tracks-in-parallel-readout. In addition, the mass memory of DBC is content-addressable. Given a cylinder number and a query conjunction, it is possible to content-search the entire cylinder 'on the fly' in the same revolution. For a detailed description of the organization of MM, the reader is referred to [2]. SFP consists of two components - the security and clustering unit (SCU) and the post processor (PP). SCU stores all the tables needed for the enforcement of security and the clustering of records and is able to search these tables in parallel. PP, on the other hand, performs the following three functions.

(1) Sorting of retrieved records. Once the response set of a user query is given to PP by the mass memory, this set of records can be sorted in the post processor.

(2) The natural and implicit join operations on two sets of records retrieved from the mass memory. These operations are particularly needed on records retrieved from a relational database.

(3) The set functions (maxima, minima, average, count, sum, set inclusion) on the response set of a user query after the records of the response set have been retrieved from the mass memory (MM) and given to the security filter processor (SFP).

As a parallel processing unit, PP is configured and interconnected with many identical processors and local memories as shown in Figure 2. This configuration is an improvement over the one presented in [4] and is the result of our recent study [5]. We note that PP consists of n processors (numbered 0 through n-1), where processor i is connected to processor (i+1) for all $0 \le i \le (n-2)$ and processor (n-1) is connected to processor 0. We shall often refer to the sequence of processors 0, 1, 2, ..., (n-1) as the broadcast sequence. We also see that PP has a post processing controller



DBCCP:	Data Base Command Control Processor
KXU:	Keyword Transform Unit
SM:	Structure Memory
SMIP:	Structure Memory Information Processor
IXU:	Index Translation Unit
MM:	Mass Memory
SFP:	Security Filter Processor
SCU:	Security and Clustering Unit
PP:	Post Processor

Figure 1. The Architecture of DBC

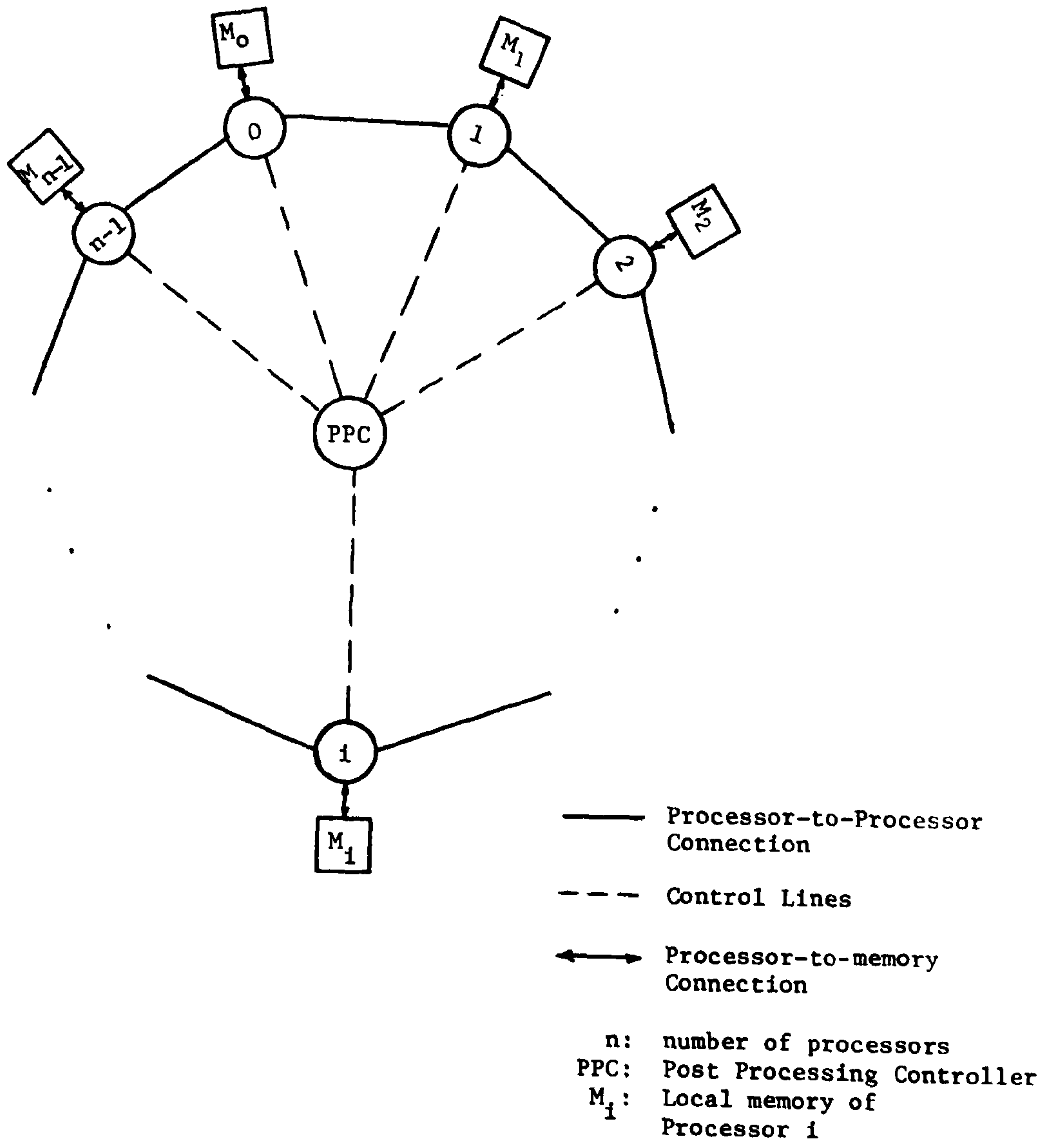

M_o
M_1
M_n-1
M_2
0
1
n-1
2
PPC
i
M_i
Processor-to-Processor Connection
Control Lines
Processor-to-memory Connection
n: number of processors
PPC: Post Processing Controller
M_i: Local memory of Processor i

Figure 2. Interconnection of Processors in the Post Processor (PP)

(PPC) to coordinate the work of the various processors in PP. PPC is directly connected to all the processors, each of which contains a small associative memory to speed up the join operation. SCU is well documented in [3], whereas PP is documented in [4, 5].

The structure loop is composed of four components: the _keyword transformation unit_ (KXU), the _structure memory_ (SM), the _structure memory information processor_ (SMIP) and the _index translation unit_ (IXU). KXU converts keywords into their corresponding internal representations. SM is primarily used to store, retrieve and update the indices of the database. Indices are maintained in SM as a directory. Each entry in the directory consists of a keyword (or keyword decriptor) followed by a set of indices. An index consists of a cylinder number and a security atom number. The cylinder number indicates where in the mass memory records containing the keyword (or derivable keywords) may be found. The security atom number indicates the security privileges accorded to records containing the keyword (or derivable keywords). SMIP is responsible for performing set intersections on the indices retrieved by SM. IXU is used to decode the indices output by SMIP. These four components are designed to operate concurrently in a pipeline fashion. The hardware organization, details and design philosophy of these components are documented in [6].

3. A REVISIT OF THE DBC METHOD OF JOIN

In order to make this paper self-contained, we repeat the definitions and notations on join operation which were developed in [4]. We will also describe and explain a modification to that operation.

3.1 Notations

Information is stored into and retrieved from DBC in terms of records; a record is made up of a collection of keywords and a record body. The record body consists of a (possibly empty) string of characters which are not used for search purposes. A keyword is an attribute-value pair, where the attribute may represent the type, quality or characteristic of the value. An example of a record with three keywords is shown below:

(<Relation, EMPLOYEE>, <Salary, 5000>, <Job, MANAGER>).

The above record has three keywords. The list of attributes constituting the above record is [Relation, Salary, Job].

Two sets of records are involved in a join operation. For a record set to participate in a join operation, it must have the following special property. Each record in the set must have the same list of attributes. The following set of two records possesses this property.

Record 1: (<Job, MANAGER>, <Name, HSIAO>).

Record 2: (<Job, SECRETARY>, <Name, ANNE>).

The list of attributes that characterizes the above set is [Job, Name]. Each attribute in the list of attributes that characterizes a record set is said to belong to that set. Thus, in the above example, Job and Name are attributes that belong to the set that they characterize. One of the sets that participates in the join operation is called the source set. The other set is called the target set. Records in a source set are called source records and records that make up the target set are called target records. A join operation is always performed between an attribute that belongs to the source set and an attribute that belongs to the target set.† We call these two attributes as the source attribute and the target attribute, respectively. Furthermore, a join operation requires that the domain of values that the

†The extension to the case when the join is performed between a list of attributes in the source set and a list of attributes in the target set is straightforward and is left to the reader.

source attribute may take be exactly the same as the domain of values that the target attribute may take.

For example, consider that the source set is characterized by the list of attributes [Name, Salary], and that the target set is characterized by the list of attributes [Salary, Department-Number]. Now, assume that the domain of the Name attribute consists of some specific alphabetical strings and the domains of the Salary and Department-Number attributes consist of some specific numbers. Therefore, a join operation may be performed between the Salary attribute belonging to the source set and the Salary attribute belonging to the target set (i.e., we choose Salary as both the source and target attributes). Similarly, another join operation may be performed between the Salary attribute belonging to the source set and the Department-Number attribute belonging to the target set (i.e., we choose Salary as the source attribute and Department-Number as the target attribute). However, no join operation is possible if we choose Name as the source attribute, because no attribute belonging to the target set has the same domain as the Name attribute. A join operation is performed as follows. Each record in the source set is examined and the value of the source attribute is determined. All records in the target set which have the same value for the target attribute are now selected. If the user is given attribute-value pairs from the selected target records, then the resulting operation by the PP is called an implicit join. However, if the user is given attribute-value pairs from both the source record used to make the selection and the target records selected, then the resulting operation is a natural join.

We shall illustrate the difference by means of an example. Consider that the source set is characterized by the list of attributes [Name, Salary], and that the target set is characterized by the list of attributes [Salary, Department-Number]. Consider the following two join operations:

(1) Join the source set and the target set using Salary as both the source and target attributes, and extract the department numbers of the target records whose salaries appear in the source records.

(2) Join the source set and the target set using Salary as the source and target attributes, and extract the name of the source records and department number of the target records and concatenate the name from a source record

and the department number from a target record if
the two records have the same salary.

We note that the first join is implicit and the second join is natural. From now on, we shall refer to source attribute values and target attribute values (which participate in the equality join) as join values.

3.2 Revising DBC Join Method

The method for doing joins has been described in [4] and requires that each processor is associated with two memories. These memories are referred to as the A-memory and the B-memory, respectively. The A-memories are used to store the source records. The B-memories are used as buffer memories in order to overcome the speed disparity between the rate at which records are being output by the mass memory (MM) and the rate at which records are being processed by the post processor (PP). The method to be described here is the same one as in [4] except that we have introduced an additional memory per processor called the C-memory. Thus, each processor has now associated with three memories. The C-memory will be used to store the results of the join operation. Also, it will be a vital component during the execution of the m-way join operation to be defined and described in Section 6.

It will be seen that the join proceeds in two distinct phases. The records in the source set are first read into the B-memories of all the n processors. That is, if there are C_s records in the source set each processor will receive C_s/n records. Each processor begins to execute the following algorithm the moment the first source record is placed in its memory. It reads the source records, one at a time, and extracts the value of the source attribute. This is then sent to PPC along with a 'request-list-#' request.

An associative memory (AM) is originally employed in PPC. Each entry in AM contains a source attribute value. Each source attribute value stored in AM has a corresponding list-#. The list-# corresponding to a source attribute value SA is i, if SA is stored in the i-th location of AM. PPC searches its AM looking for an entry which is equal to the source attribute value passed to it. If such an entry is found, the corresponding list-# is returned to the processor which issued the 'request-list-#'. Otherwise, a new entry is made in AM for this source attribute value and the corresponding list-# is returned to the processor which issued the 'request-list-#'.

The processor now hashes the list-# into a block of its A-memory. The record is then stored in this block of the A-memory, if space is available in the block. Otherwise, it is stored in an overflow area. A few locations in each processor's A-memory are reserved to store overflow records. Each record in the overflow area contains a pointer to the next record in the overflow area that hashed to the same block. One memory location in each block is reserved for an overflow pointer. The overflow pointer contains a pointer to the first overflow record in the overflow area that hashed to this block.

At the end of this phase of the join operation, all the source records have been placed in appropriate blocks of the A-memories depending on their source attribute values. Thus, in this phase, the B-memories are used as input buffers whereas the A-memories are the intended store of the source records.

Next, the target records are read in a similar manner into the B-memories. Each processor begins to execute the following algorithm the moment the first target record is placed in its memory. It reads the target records in its memory one at a time and extracts the value of the target attribute. This value is then sent to PPC which checks AM to see if this value can be associated with any list-#. If so, it returns the corresponding list-#. Otherwise, it returns a null value for the list-#. The processor then hashes the non-null list-# to a block address. The hash algorithm used here is the same one that was used earlier to hash the join values of source records. All the source records in this block of the A-memory and all overflow records belonging to this block are now looked at. If the join value is the same as the join value of the target record that was hashed to this block, then this source record will participate in the join operation. The necessary attribute values are extracted from both the source and target records and placed in the C-memory. The results are later sent to DBCCP which will then route them to the front-end computer from which the request originates. Again, for the target records, the B-memories serve as the input buffer. The C-memories are used as the output buffer.

If the list-# corresponding to the join value of the target record was non-null, this record and the list-# corresponding to its join value are propagated to all other processors via the broadcast sequence. That is, each processor sends the record to one other processor -- the one following it in the broadcast sequence. In this way, the record is propagated to every other processor. The last processor in the sequence will discard the record.

When it receives a broadcasted record, each processor treats the record like any other record received directly from MM by way of the B-memory, except for the fact that the AM need not be accessed to determine the list-# corresponding to the join value since this is propagated along with the record.

We note that the associative memory (AM) is used to speed up the join operation. Since our method is based on a hashing scheme, the use of the AM in a join operation eliminates from consideration all those target records which will not participate in the join operation. Without the AM, we will need to look for and process every target record.

4. ANOTHER MODIFICATION TO THE JOIN OPERATION

In addition to the use of C-memories, we would like to propose a variation in the use of the associative memory (AM). Presently, the AM is a part of the post processing controller (PPC) and must be large enough to hold all unique source attribute values extracted from the source records. Every processor in PP needs to access this AM to determine if certain target records will participate in the join. This leads us to the following observations:

First, since all of the processors must access this common AM, an increase of the number of processors in PP will require an increase of AM access speed. Thus, each time that we increase the number of processors in PP, we may have to replace the present AM with a faster one. Otherwise, the AM will become a bottleneck. The above discussion implies that the PP is not easily hardware-extensible.

Secondly, since the AM stores the source attribute values of all source records, it must be fairly large. This would require that we use fully associative memories with large capacity in order to make sure that the AM is fast enough to search through this fairly large set of values. Fully associative memories, such as STARAN, of large capacity and high speed are expensive. We therefore propose that instead of one common, large and fast AM, each processor is allowed to have its own, small and slow associative memory (am). The am attached to a processor will store only all the unique source attribute values of the source records present at that processor. Thus, if there are n processors in PP, each individual am will be approximately $\frac{1}{n}$ as large as the common AM. That is, the total size of all the ams will be equal to the size of the common AM in the previous design. In summary, the use of ams has the following advantages:

(1) Each am is much smaller in size than the common AM. Specifically, the size of am is approximately equal to $\frac{1}{n}$ x (size of AM). This means that it may now be possible to realize the am with a random-access memory (RAM) and a more conventional search.

(2) Since each processor will access only its local am, each am needs to respond to requests at a much slower rate than the common AM has to in the previous design. This is because the common AM has to respond to requests from all n processors rather than only one processor. In fact, each am requires only to be $\frac{1}{n}$ times as fast as the common AM in the previous

design.

(3) The PP is now hardware extensible. This is because the addition of new processors does not place an additional burden on each am. In fact, the load is probably reduced, since the existing ams now need to store the unique source attribute values from a smaller number of source records.

The suggested change thus seems to be superior in all respects. There is, however, one additional point that remains to be considered. We want to ensure that the substitution of a single large AM with several smaller ams does not affect the time complexity of the join operation. Accordingly, we make a simple analysis of the time complexities of the join operation for both cases. The analysis is kept deliberately simple in order not to clutter up our comparison of complexities with unnecessary details. The complexity results are presented in terms of the cardinalities of the source, result and target sets. In [4], we had analyzed the time complexity in terms of the sizes of the A-memories. However, we now feel that a more meaningful complexity measure would be in terms of set cardinalities.

4.1 Analysis for the Case Where a Common Associative Memory (AM) is Used

The following notations are introduced to simplify the ensuing discussion:

C_s	:	Cardinality of the source set.
C_t	:	Cardinality of the target set.
C_r	:	Cardinality of the result set.
Z_a	:	Average time to access and read (write) a record from (to) the A-memory.
Z_{bs}	:	Time taken to read (write) next record from (into) the B-memory.
Z_{AM}	:	Time taken to probe the AM.
Z_{am}	:	Time taken to probe an am.
N	:	Number of blocks (buckets) in the A-memory.
e	:	Efficiency of the hashing function [e.g., $e = 1$, if the hashing scheme is perfect and there are no collisions].
n	:	Number of processors in PP.
a	:	Fraction of target records that participate in the join operation.

We will analyze each phase of the join operation, in turn.

4.1.1. The Phase - I Analysis:

In this phase, the following operations are performed by a processor on each source record in its B-memory. The record is read from the B-memory, its join value is stored in the AM (if the join value is not already there), it is hashed to a block of the A-memory and the record is stored in that block of the A-memory. We will include the hashing time in Z_{AM} in our calculations. Thus, the processor spends $(Z_{bs} + Z_{AM} + Z_a)$ time units in servicing a single record. Since each processor has $\frac{C_s}{n}$ records, the total time taken for this phase is

$$\frac{(Z_{bs} + Z_{AM} + Z_a)C_s}{n} \quad . . . \quad (I)$$

4.1.2. The Phase - II Analysis:

Each processor receives $\frac{C_t}{n}$ target records directly from the mass memory (MM). Each of these records is read from the B-memory and then the join value of the record is checked against the list of join values in the AM. This takes the following amount of time

$$\frac{C_t}{n}(Z_{bs} + Z_{AM})$$

After checking each join value against the list of join values in the AM, it will be seen that only a fraction, a, of the target records will participate in the join operation. That is, $\frac{aC_t}{n}$ records will participate in the join operation. Each of these records will be hashed to a block of the A-memory and all the records in that block must be accessed and checked for possible participation in the join operation. Since there are $\frac{C_s}{n}$ source records in the A-memory of a processor and the A-memory has N blocks, we expect that each block will contain $\frac{C_s}{nN}$ records. Thus, this process takes the following amount of time

$$\left(\frac{aC_t}{n}\right)\left(\frac{C_s}{Nn}\right) Z_a, \text{ i.e., } \frac{aC_tC_sZ_a}{n^2N}$$

In addition to the $\frac{C_t}{n}$ target records received directly from the MM, each processor also receives $\frac{a(n-1)C_t}{n}$ records from the other processors. Each of these records is read from the B-memory, hashed to a block of the A-memory, and all the records in that block are read out for possible participation in the join operation. Once again, we expect that each block will contain $\frac{C_s}{nN}$ records. Thus this takes the following amount of time

$$\frac{a(n-1)C_t}{n}(Z_{bs} + \frac{C_s Z_a}{nN})$$

Finally, we should also include the time taken to route records from one processor to the next one in the broadcast sequence. However, we make the assumption that such routing of records from one processor to (the B-memory of) the next processor is done by the use of a direct memory access (DMA) device so that the routing is overlapped with record processing. Hence, the record routing time is ignored in our calculations. The total time for the phase - II is

$$\frac{C_t}{n}(Z_{bs} + A_{AM}) + \frac{(n-1)aC_tZ_{bs}}{n} + \frac{aC_tC_sZ_a}{nN} \quad . \; . \; . \quad (II)$$

Combining formulae I and II, we arrive at the following expression for the time to do join using a common AM.

$$\frac{C_s}{n}(Z_{bs} + Z_{AM} + Z_a) + \frac{C_t}{n}(Z_{bs} + Z_{AM}) + \frac{(n-1)aC_tZ_{bs}}{n} + \frac{aC_tC_sZ_a}{nN} .$$

By looking at the time-complexity expression derived above, we note that if the fraction of target records that participates in the join is in the range of 1% to 10%, then the third term will likely be the smallest and may be negligible. In that case, the time complexity is of the order

$$O(\frac{C_s}{n} + \frac{C_t}{n} + \frac{aC_tC_s}{nN})$$

We now see that each of the three terms has n (the number of processors) in the denominator. Thus, by using n processors in PP, this join algorithm will be n times as fast as with a single processor in PP, and this is the best performance that can be expected from a parallel organization of n processors.

We note that the last term in the time-complexity expression is a product of the source and target cardinalities and we must reduce the effect of this term as much as possible. First, we note that this term is also multiplied by the fraction, a, which is typically very small. Next, it is divided by the number of buckets, N, in the A-memory. Thus, the effect of this term may also be offset by an increase in the number of buckets in the A-memory.

In conclusion, we have a join method that performs approximately n times faster with n processors and has a complexity which is the product of the source and target set cardinalities. Can we possibly do better by using multiple associative memories each of which has a smaller capacity and slower speed?

4.2 Analysis for the Case Where Multiple Associative Memories (ams) are Used

Once again, we analyze each phase separately.

4.2.1 The Phase - I Analysis

In this phase, the following operations are performed by a processor on each source record in its B-memory. The record is read from the B-memory and its join value is stored in the am (if not already there). The join value is then hashed to a block of the A-memory and the record is stored in that block of the A-memory. Thus, the processor spends $(Z_{bs} + Z_{am} + Z_a)$ time units in servicing a single record. Since each processor has $\frac{C_s}{n}$ records, the total time taken for Phase - I is

$$\frac{(Z_{bs} + Z_{am} + Z_a)C_s}{n} .$$

4.2.2 The Phase - II Analysis

Each processor receives $\frac{C_t}{n}$ target records directly from the mass memory (MM) and $\frac{(n-1)}{n}C_t$ records from the other processors in the PP. Each of these records is read by the processor from its B-memory and then the join value of the record is checked against the list of join values in the ams. This takes the following amount of time

$$C_t(Z_{bs} + Z_{am}) .$$

Now, those target records which may participate in the join operation (as indicated by the ams) are hashed to blocks of the A-memory and result records are created.

Let us assume that each processor generates $\frac{C_r}{n}$ of the result records. If the hashing scheme used were perfect (i.e., collision free), then the number of accesses to the A-memory should exactly be equal to the number of result records generated $(\frac{C_r}{n})$, since each access of the A-memory will retrieve a record that will participate in the join operation and produce a result record. This is because the use of the am has eliminated from consideration those target records that will not produce result records. Thus, this operation will take the following time

$$\frac{C_r Z_a}{n} .$$

However, if the hashing scheme used were imperfect with efficiency e, then this operation will take the following time

$$\frac{C_r Z_a}{ne} .$$

Hence, the total time for the Phase - II is

$$C_t(Z_{bs} + Z_{am}) + \frac{C_r Z_a}{ne} .$$

The total time for the entire join operation is

$$\frac{C_s(Z_{bs} + Z_{am} + Z_a)}{n} + C_t(Z_{bs} + Z_{am}) + \frac{C_r Z_a}{ne} ,$$

$$\text{i.e., } O(\frac{C_s}{n} + C_t + \frac{C_r}{n})$$

The first observation we wish to make regarding the join using multiple ams is that the time is linear in the cardinality of the source, target and result sets. We postulate that no join algorithm can do better than this for the following reasons. Since we need to read in all the source and target records, the input process has to be linear in the cardinality of the spurce and target sets. Also, each record in the result set must be created and output, and this creation and output process must be linear in the cardinality of the result set. Thus, the entire process of join has to be linear in the cardinalities of the source, target and result sets. The method proposed in [13], for example, has the same time complexity as our method although it uses a costlier and more complicated architecture.

The second observation we make is that if the target set, C_t, is very small relative to $\frac{C_s}{n}$ and $\frac{C_r}{n}$, we may drop this term to obtain the time complexity as

$$O(\frac{C_s}{n} + \frac{C_r}{n}).$$

Another reason for dropping the middle term containing C_t is because C_t is multiplied by Z_{bs} and Z_{am} the access times of the B-memory and the associative memory, both of which are fast memories. On the other hand, the term containing C_s and the term containing C_r are both multiplied by Z_a, the access time of the much slower A-memory. From the above complexity, we observe that the join algorithm will perform n times faster if we use n processors in PP rather than a single processor, and this is the best that can be expected.

4.3 Concluding Remarks on the Analysis

Analysis of the time complexities of the join operation in the two cases indicates that the method using slower and smaller associative memories (ams) is superior to the method using a single common and fast associative memory (AM). In fact, we feel that the join operation using multiple ams should perform as well as or better than any other method for doing join.

In an earlier section, we had indicated three points which illustrated the superiority of the multiple am method over the single AM method. This section indicates one more reason for our claim that the multiple am join is superior to the single AM join. We conclude by saying that the suggested change will be incorporated into the design of the PP.

5. QUEUEING ANALYSIS OF JOIN OPERATION

In this section, we wish to analyze the algorithm described in the previous section in order to arrive at reasonable estimates for the sizes of the various memories. We will make no attempt to estimate the size of the C-memory associated with each processor. We merely state that it must be large enough to hold the results from any join operation. Reasonable estimates for the sizes and speeds of the A and B-memories are, however, obtained. We first analyze the algorithm for the case when n, (i.e., the number of processors) is 1. Such an analysis is made for two reasons. First, since no interprocessor communication is involved, the calculations are simpler than for the case where $n > 1$. Second, certain results obtained for the case where $n = 1$ may be used to simplify the calculations in the case where $n > 1$.

The analysis of the A and B-memory sizes is made under the assumption that no associative memories (either AM or ams) are used to speed up the join. Thus, the result can be used as the lower bound of the speed of the processor and memory organization for the post processor (PP).

5.1 Analysis for the Single-Processor Case

For the single-processor case, all the records in the source set are first read into the B-memory and then stored in the A-memory. Next, the records in the target set are read into the B-memory. The moment the first target record is placed in the B-memory, the processor begins to execute the following algorithm.

It reads the target records one at a time, hashes the join value of the target record to a block in the A-memory, reads all source records in this block (henceforth, whenever we refer to records in a block, we mean all records in the block plus all records in the overflow area that hashed to the same block) and performs the join on all relevant records. The target record may now be discarded, and the next target record read. Thus, there is a certain rate at which records arrive to the B-memory (disk parallel read-out rate), and there is a certain rate at which these records are serviced. It should thus be possible to analyze the size of the B-memory based upon these arrival rates and service rates. In the following sections, we shall analyze the sizes of the A and B-memories.

5.1.1 An Analysis of the A-Memory Size

As stated earlier, the A-memory consists of a primary area, and an overflow area. The primary area consists of N blocks, each of size R (where $N \times R = C_{smax}$, the cardinality of the largest possible source set). If more than R records are assigned to a block, the excess records are stored in the separate overflow area common to all blocks. For each block with overflow records, there is, in the overflow area, a chain of overflow records with address pointers. The overflow area must be large enough to hold the worst-case overflow which will occur when all C_{smax} records hash to the same block. In this case, the overflow area must be large enough to hold $(C_{smax} - R)$ records and as many address pointers. Ignoring the size of address pointers with respect to the size of records, the A-memory must be large enough to hold the following records

$$\begin{aligned} & N \times R + C_{smax} - R \\ = {} & C_{smax} + R(N-1) \end{aligned}$$

5.1.2 An Analysis of the B-Memory Size

From our discussion in an earlier section, it is quite clear that the B-memory contains a queue of records waiting to be processed by the post processor (PP). Thus, we can arrive at a formula for the B-memory size by estimating the number of records in this queue. In order to estimate the size of this queue of records waiting to be processed (and hence, the size of the B-memory), we will make use of well-known results from the queueing theory.

In [7], a simple shorthand notation, A/B/m, is used for describing queues, where m is the number of servers (in our case the processor is the server) and A and B indicate the distributions of the interarrival time and service time, respectively. For our analysis, the interarrival time for the queue of records in the B-memory is considered to be <u>expontially distributed</u> and the service time of this queue is <u>generally distributed</u>. The rationales are the following: Consider the interarrival distribution. We recall that the records must first be selected from the mass memory (MM) and only those records that are selected for participation in the join will arrive at the post processor. Since a record is selected by the MM for the join independent of whether any other record is selected for the join, the arrival time of a particular record at the post processor is also independent of the number, rate and time of previous record arrivals. Thus, the record arrival rate is exponential which is known to model a continuous distribution of independent arrivals [7; pp. 65-67]. On the other hand, the service time distribution is not exponential. This is because the amount of service that a record continues to receive will depend upon (1) how much service it has already received and (2) how much service the previous record had received. The service time distribution is not Erlangian because it cannot be represented as a number of exponentially distributed service times. Instead, the service time distribution is rather arbitrary which is termed general [7; pp. 3,4]. Thus, the queue of records being served with one processor is an M/G/1 queue where M designates exponential interarrival time distribution, G denotes general service time distribution and 1 represents one server. The queue is shown in Figure 3. We are now in a position to use the result for the average queue length of an M/G/1 queue from [7; pp. 15,16]. Let S_B be the average number of records in queue (i.e., the average number of records in the B-memory or the average size of the B-memory.) Then, we have

$$S_B = \delta + \frac{\lambda^2\overline{x^2}}{2(1-\delta)}$$

where,

λ = E(arrival rate)

$\bar{x}$ = E(service time)

$\overline{x^2}$ = E(service time^2)

and $\delta = \lambda\bar{x}$

[Note: We use the notation E(P) to stand for the expected value or mean of a random variable P].

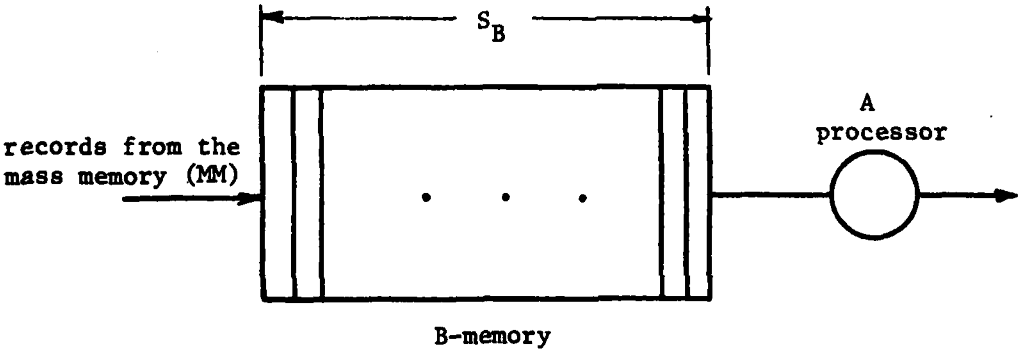
In Appendix 1, we calculate λ, $\bar{x}$, $\overline{x^2}$ and δ for the queue of records in the B-memory. We now have an equation for the size of the B-memory. This equation gives the size of the B-memory in terms of various parameters such as the access times of the A and B-memories, and the size of the A-memory. Substituting typical values for these parameters will allow us to arrive at typical values for the B-memory size. However, instead of doing so, we shall proceed to analyze the PP when it consists of more than one processor since this is the more interesting case.

arrival distribution $= \frac{p}{r} e^{-\frac{p}{r} \cdot t}$

Mean inter-arrival time $= \frac{r}{p}$

variance of inter-arrival time $= \frac{r^2}{p^2}$

C_s, p, r, y, R, N, Z_{as}, Z_{bs}, S_B and Z_{ar} are defined in Section 5.1.2 as well as in Section 5.3. v and Q are defined in Appendix 1.



Mean service time =

$$Z_{bs} + (Z_{ar} - Z_{as}) \cdot \frac{y}{v} \cdot \frac{PC_s}{N} + \frac{Z_{as} PC_s}{N} (1 - Q(R))$$

$$+ Z_{ar} \cdot Q(R) \cdot \frac{PC_s}{N} + R\,(Z_{as} - Z_{ar}) \cdot Q(R+1)$$

Variance service time = as in Appendix 1

Figure 3. The M/G/1 Queue of Records in the B-memory for the Uniprocessor Case

5.2 An Analysis For the Multi-Processor Case

The algorithm, as has been explained before, is as follows. First, all the records in the source set are hashed into the A-memories of all the processors. If there are C_s records in the source set, each processor will have $\frac{C_s}{n}$ records at the end of this phase of the join. Next, the records in the target set are read into the B-memories. Each processor performs the following the moment the first target record is placed in its B-memory. It reads the target records in its memory one at a time and hashes the join value of the record (using the am) to a block of the A-memory. All records in this block of the A-memory (including overflows) are read and the join is performed on all relevant records. The target record in the B-memory is now propagated to all the other processors in the broadcast sequence. The record is discarded after it has propagated through all the n processors. Once again, we shall analyze the size of both the A- and the B-memories associated with each of the n processors.

5.2.1 An Analysis of the A-Memory Size

The analysis here is analgous to the analysis for the uniprocessor case. Each processor's A-memory consists of a primary area, and an overflow area. The primary area consists of N buckets, each of size R. If more than R records are assigned to a bucket, the excess records are stored in the separate overflow area common to all buckets. For each bucket with overflow records, there is, in the overflow area, a chain of overflow records with address pointers. The overflow area must be large enough to hold the worst case overflow which will occur when all $\frac{C_{smax}}{n}$ source records hash to the same bucket. In this case, the overflow area must be large enough to hold $(\frac{C_{smax}}{n} - R)$ records and as many address pointers. Ignoring the size of address pointers with respect to the size of records, the A-memory must be large enough to hold

$$(N \times R + \frac{C_{smax}}{n} - R) = \frac{C_{smax}}{n} + R(N-1) \text{ records}$$

5.2.2 An Analysis of the B-Memory Size

We are interested in analyzing the size of the B-memory attached to a processor. That is, we are interested in estimating the number of records that are in the queue of a single processor, waiting to be processed by that processor.

Figure 4 shows the model used to make our analysis. The analysis is more complicated now than for the single processor case because the interarrival time distribution is no longer exponential. As can be seen, there are two factors that contribute to the arrival rate of records to the B-memory. The first is the arrival of records directly from the mass memory (MM). The second factor is the arrival rate of records from the previous processor in the broadcast sequence. Thus, the interarrival distribution depends upon the service time distribution (which is general, as for the uniprocessor case). In Appendix 2, we find that the interarrival time distribution and the service time distributions are both general as intuitively expected. Thus, in the terminology of [7], the queue of records waiting to be processed by a processor is a G/G/1 queue. In order to calculate the queue length of a G/G/1 queue, we need to calculate the first and second moments of the interarrival time distribution and the service time distribution. These are calculated in Appendix 2. The queue is shown in Figure 5. Exact solutions to the average queue length of a G/G/1 queue are unknown. Hence, in analyzing our queue, we will have to use approximations available in the literature. In Appendix 3, we derive an expression for the average queue size using the Heavy Traffic Approximation [7]. In Appendix 4, we derive an expression for the average queue size using the Diffusion Approximation [10]. We also derive, in the appendices 3 and 4, two expressions for the size of the B-memory attached to a processor in order to be 95% certain that the B-memory will not overflow. These two expressions are derived by using the two different approximation methods.

In the next section, we shall utilize the expressions to calculate the actual sizes of the A and B-memories.

5.3 Interpretation of the Results on Memory Size

We are now able to use the expressions derived in section 5.2.1 and Appendices 3 and 4, to arrive at the sizes of the A and B-memories in terms of actual numbers. We also want to handle the worst possible cases. This will occur when the source set is very large and when all the source records are selected for the join operation. Accordingly, we let

C_s = the cardinality of the source set = 100,000 records.

p = the fraction of source set records selected for the join operation = 100%.

q: fraction of records not discarded by a processor after servicing
D: average arrival rate of records from mass memory
A: average service rate of records

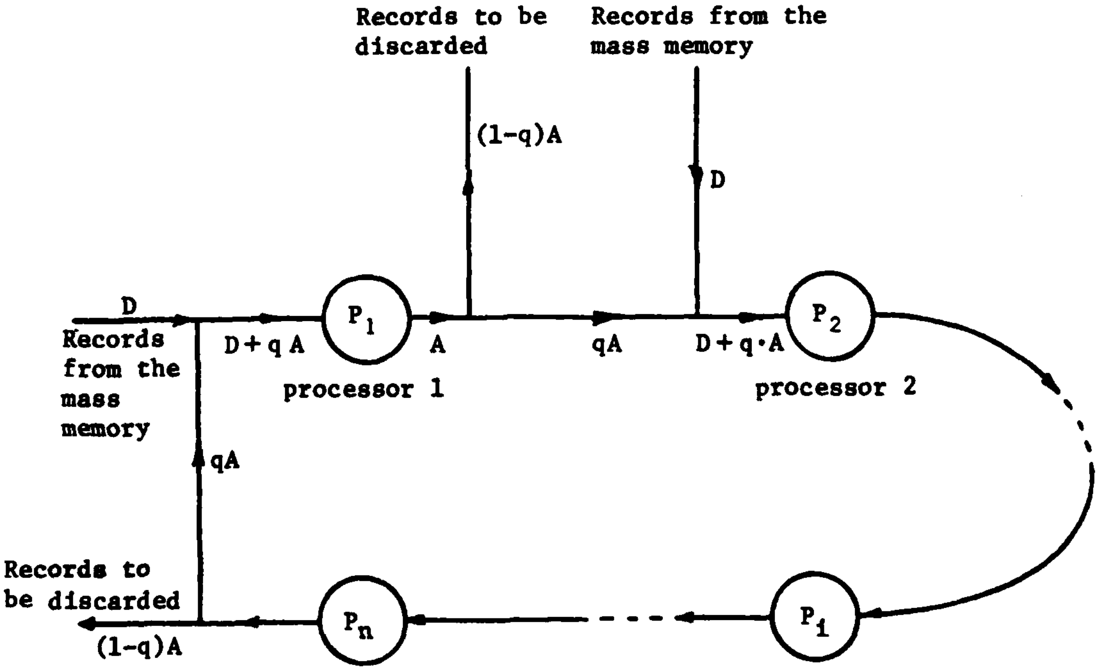


Figure 4. Model Used for Developing the Inter-arrival Time Distribution of Records to the B-memory in the Multiprocessor Case

q is defined in Figure 4.

C_s, p, r, y, R, n, N, Z_{as}, Z_{bs} and Z_{ar} are defined in Section 5.3,

S_B and δ are defined in Section 5.1.2

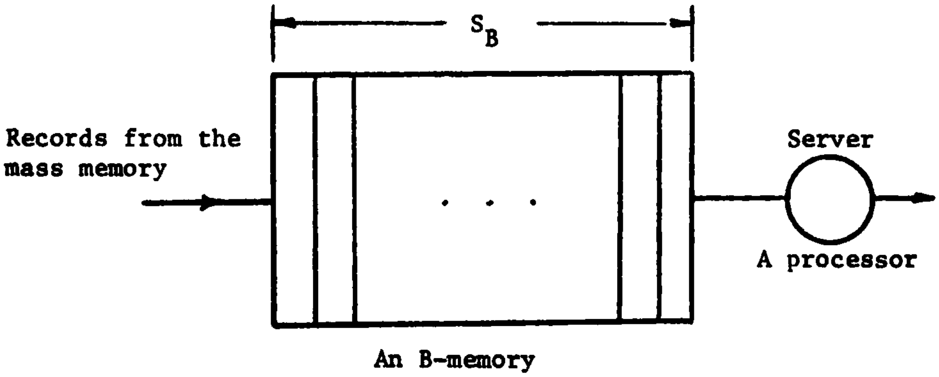
Q and v are defined in Appendix 1.

$\propto$(service time) is defined in Appendix 2.

Mean service time =

$$Z_{bs} + Z_{ar} - Z_{as} + \frac{y}{v}\,\frac{pC_s}{nNOB}$$
$$+ Z_{as}\frac{pC_s}{nN}(1-Q(R)) + Z_{ar}\frac{pC_s}{nN}Q(R)$$
$$+ R\,(Z_{as} - Z_{ar})Q(R+1)$$

Variance of service time is as in Appendix 2



Mean inter-arrival time $= \dfrac{(1-q)\cdot nr}{p}$

$$\text{Variance} = \frac{(1-q)^2n^2r^2}{p}\left[\frac{q^3\cdot\delta^2\cdot\propto(\text{service time})}{1-q^3(1-\delta^2)} + 1\right]$$

Figure 5. The G/G/1 Queue of Records in the B-memory for the Multiprocessor Case

r = the inter-arrival time of records from MM at maximum read-out rate = 50 μsecs [This is arrived at in the following manner. In DBC, an average record is 500 bytes. For the MM disk, we consider the Ampex Parallel Transfer disk [11] which stores 20,000 bytes per track and can read out of 9 tracks in parallel. Assuming a disk rotation speed of 3,600 rpm, we obtain the time to read one record as

$$= \frac{1}{3600} \times 60 \times \frac{500}{20{,}000} \times \frac{10^6}{9} \ \mu\text{secs}$$

$$= 46.3 \ \mu\text{secs} \simeq 50 \ \mu\text{secs}].$$

y = the time taken to join two records = 50 μsecs.

R = the number of records per bucket of the A-memory = 10. [The average CCD segment size is 5 kbytes and a DBC record is approximately 500 bytes].

n = the number of processors [chosen from the set of 16, 32, and 48].

N = the number of buckets [chosen from the set of 10^4, 10^5 and 10^6].

a = the fraction of target records that participate in the join operation [chosen from the set of .2, .5 and 1].

z_{as} = the time to read a record (500 bytes) sequentially from a given block of the A-memory once the block is found. [In other words, the block access time is not included. Once again, assuming that the A-memory is made of CCDs, z_{as} is chosen from the set of 5 μsecs, 10 μsecs, 15 μsecs, 20 μsecs, 50 μsecs and 100 μsecs].

z_{bs} = the time to sequentially read out a record from the B-memory. [We note that the B-memory is not block-oriented, e.g., made of RAMs. We choose z_{bs} from the set of 10^{-9}, 10^{-8}, 10^{-7}, 10^{-6}, 10^{-5} and 10^{-4} secs].

z_{ar} = the time to access a given block in which the record resides and the time to read the record sequentially from the block. [For a CCD memory, the time taken to access an arbitrary block is about 100 μsecs. However, clever arrangement of the CCD memory organization can reduce this time to about 10 μsecs. Furthermore, the read-out time is as suggested for z_{as}. Thus, we choose z_{ar} from 15 μsecs to about 200 μsecs].

5.3.1 Results on the A-Memory Size

In Section 5.2.1, we have expressed the A-memory size in terms of various parameters such as cardinality of the source set, number of processors in PP, number of buckets in the A-memory, and so on. In this section, we provide typical values of these parameters for the expression to obtain some concrete numerical results. In Figure 6, we have tabulated the A-memory size for various values of n (the number of processors) and N n: number of processors →

N: Number of Buckets ↓

n N	1	2	5	10	20	50
500	105,000	55,000	25,000	15,000	10,000	7,000
1,000	110,000	60,000	30,000	20,000	15,000	12,000
2,000	120,000	70,000	40,000	30,000	25,000	22,000
5,000	150,000	100,000	70,000	60,000	55,000	52,000
10,000	200,000	150,000	120,000	110,000	105,000	102,000
20,000	300,000	250,000	220,000	210,000	205,000	202,000
50,000	600,000	550,000	520,000	510,000	505,000	502,000
100,000	1,100,000	1,050,000	1,020,000	1,010,000	1,005,000	1,002,000

C_s = source set cardinality = 100,000

R = number of records per block = 10

Figure 6. A-memory sizes for various values of n (number of processors) and N (number of buckets)

(the number of buckets in the A-memory). In arriving at these values, we assumed that the cardinality of the source set was 100,000 records, and that ten of these records will fit into a bucket of an A-memory.

From the tabulation, we observe that the size of the A-memory is reduced when large number of processors and small number of buckets are utilized. For example, the smallest A-memory size of 7000 records occurs when there are as many as fifty processors in PP and as few as 500 buckets in the A-memory. Intuitively, we may interpret this as follows. The A-memory as we recall, consists of a primary area and an overflow area. By decreasing the number of buckets, we are decreasing the size of the primary area which is directly proportional to the number of buckets. By increasing the number of processors, on the other hand, we are decreasing the size of the overflow area, since the possibility of bucket overflows is minimized. Thus, increasing the number of processors and decreasing the number of buckets both contribute to a decrease in the A-memory size. The minimum A-memory size should occur when the number of processors is very large and the number of buckets is very small.

Although it is true that increasing the number of processors in PP will cause a decrease in the size of the A-memory attached to a processor, the total size of all the A-memories may still go up because we now have more processors, each with its own A-memory. Also, increasing the number of processors will increase the cost of the PP in terms of processor cost. Thus, it is not enough to consider the size of A-memory in isolation. We need to consider both the A-memory and the number of processors together.

We have also seen that decreasing the number of buckets causes a decrease in the A-memory size. However, in the next subsection, we will see that decreasing the number of buckets will contribute to an increase in the B-memory size. Thus, a trade-off will have to be made for the number of buckets in the A-memory. Once again, we are lead to the conclusion that the A-memory cannot be considered in isolation. It is necessary to consider both A and B-memories together.

In a later section, we will present an integrated study of the A-memory, the B-memory and the number of processors. However, we will present an isolated study of the B-memory for the time being.

5.3.2 Results on the B-Memory Size

Now, we present the results for the size of the B-memory attached to a processor. A typical set of results for the case when the time to access a block and read a random record from A-memory is 55 μsecs and the time to read the next record sequentially from A-memory is 10 μsecs is presented in Figure 7 and also in Table I. In Figure 7a, we plot the variation of the B-memory size as a function of the speed of the B-memory for various values of n (the number of processors) and N (the number of buckets in the A-memory). As expected, the faster the B-memory, the smaller it needs to be. For example, when there are 32 processors and the A-memory has 10,000 buckets, we see that if we can read a record out of the B-memory in 10^{-6} seconds, then the B-memory has to be large enough to hold fifteen records. On the other hand, if we use a faster B-memory from which we are able to read out a record in 10^{-7} seconds, then the B-memory need only be large enough to hold eight records. One important feature of all the curves in Figure 7a is that at faster memorv speed they all become parallel to the x-axis, that is the B-memory speed axis. This means, of course, that increasing the B-memory speed (or decreasing the record read-out time) beyond a certain point will have no effect on the size of the B-memory. Thus, it is clearly not worthwhile to increase the speed of the B-memory beyond a certain point. Figure 7a suggests that decreasing the read-out time below 10^{-7} seconds is not worthwhile since it has a negligible impact on the B-memory size. On the other hand, all the curves in Figure 7a eventually become parallel to the y-axis, that is the B-memory size axis at a B-memory read-out time of 10^{-5} seconds. This means that if the time to read a record out of B-memory is 10^{-5} seconds or more, the size of B-memory has to be arbitrarily large. This means that the PP is not able to service the records in the queue (B-memory) fast enough because the read-out rate of the B-memory is too slow. In conclusion, we would require that the read-out time of a record from the B-memory be between 10^{-5} and 10^{-9} seconds. For a reasonably small B-memory size, we will require the read-out time of a record from the B-memory to be between 10^{-6} and 10^{-8} seconds.

In Figure 7b, we plot the B-memory size as a function of the number of buckets in the A-memory for various values of n (the number of processors) and Z_{bs} (the read-out time of the B-memory). As we increase the number of buckets in the A-memory, the need for a large B-memory

Z_{ar} = 55 μsecs, Z_{as} = 10 μsecs, a = 1

Z_{ar} : block access and record read-out time from A-memory

Z_{as}: record read-out time from A-memory

n: number of processors

N: number of buckets in A-memory

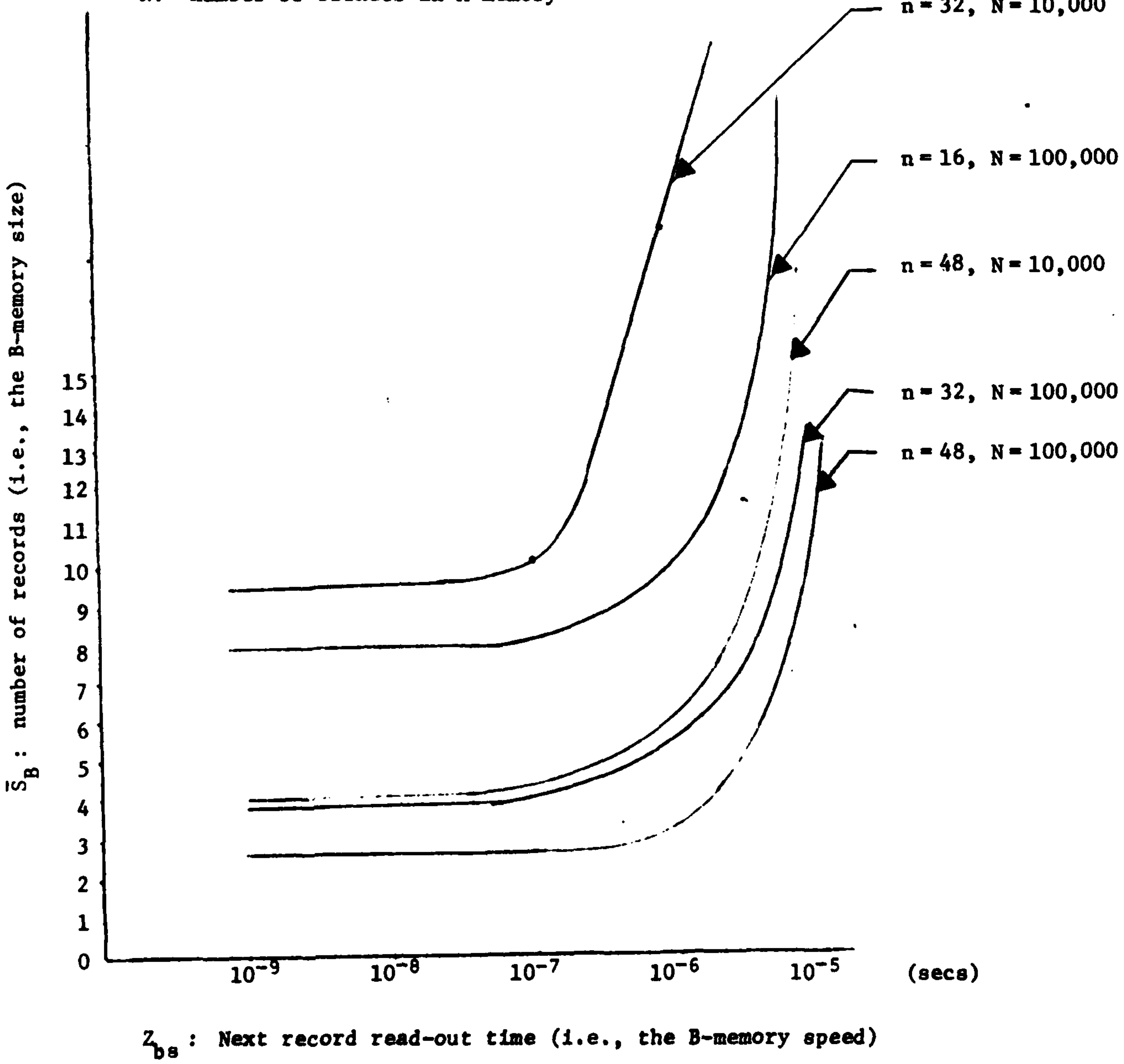


Figure 7a

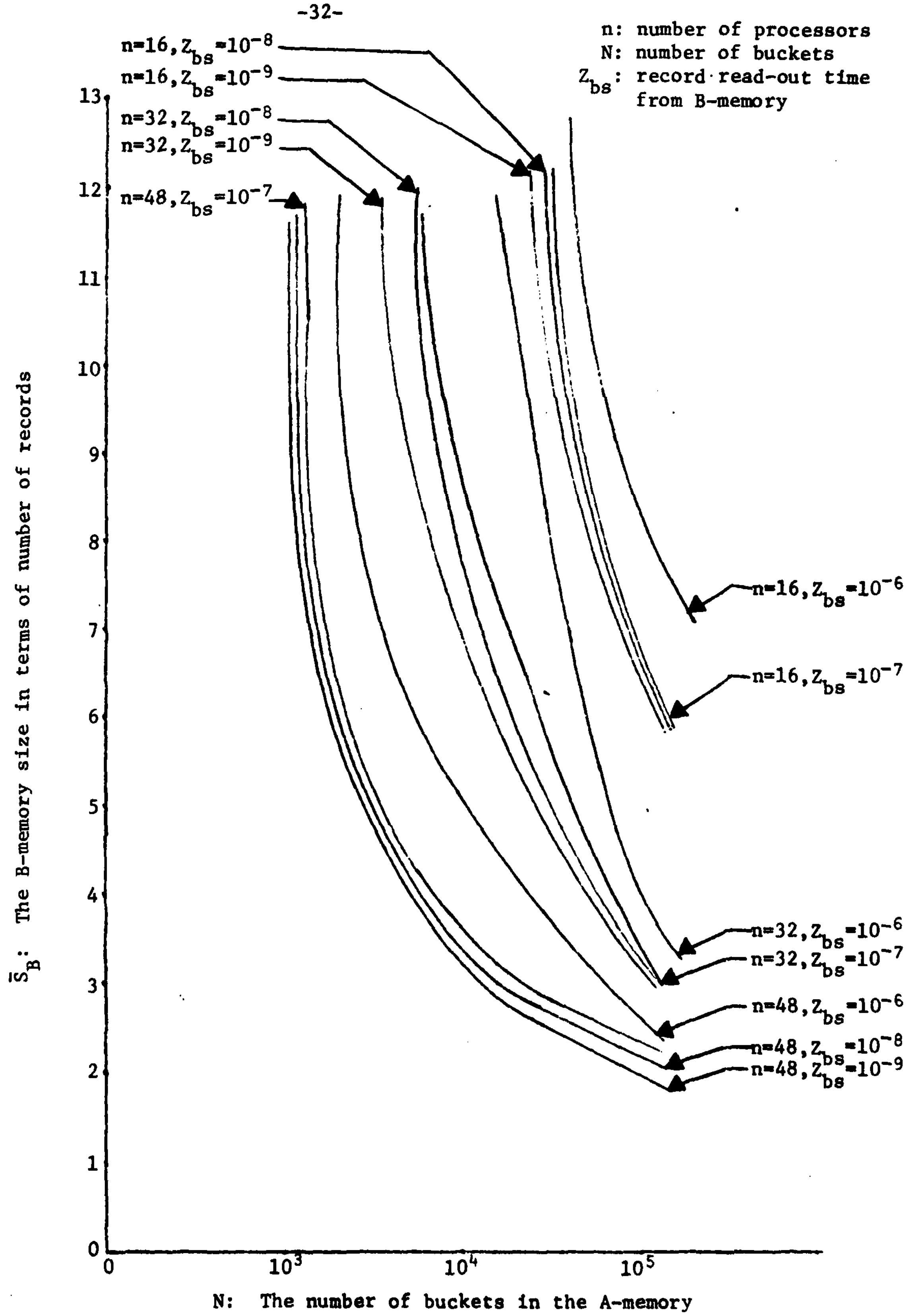

n: number of processors
N: number of buckets
Z_{bs}: record·read-out time from B-memory
$n=16, Z_{bs}=10^{-8}$
$n=16, Z_{bs}=10^{-9}$
$n=32, Z_{bs}=10^{-8}$
$n=32, Z_{bs}=10^{-9}$
$n=48, Z_{bs}=10^{-7}$
$n=16, Z_{bs}=10^{-6}$
$n=16, Z_{bs}=10^{-7}$
$n=32, Z_{bs}=10^{-6}$
$n=32, Z_{bs}=10^{-7}$
$n=48, Z_{bs}=10^{-6}$
$n=48, Z_{bs}=10^{-8}$
$n=48, Z_{bs}=10^{-9}$
$\bar{S}_B$: The B-memory size in terms of number of records
13
12
11
10
9
8
7
6
5
4
3
2
1
0
0
10^3
10^4
10^5
N: The number of buckets in the A-memory

Figure 7b

decreases. We observe that the figure has twelve curves, four corresponding to the case when the number of processors is 16, four corresponding to the case when the number of processors is 32, and four corresponding to the case when the number of processors is 48. The interesting point to note is that the curves corresponding to 16 processors are to the extreme right, the curves corresponding to 32 processors are in the middle and the curves corresponding to 48 processors are to the extreme left. Thus, with only 16 processors, the B-memory size approaches infinity unless there are 20,000 or more buckets in the A-memory. That is, if there are less than 20,000 buckets in the A-memory, the PP will not be able to service the records in the queue (i.e., in the B-memory) fast enough. With 32 processors, the B-memory size approaches infinity unless there are 5000 or more buckets. Finally, with 48 processors, the B-memory size approaches infinity unless there are 1,000 or more buckets. Thus, if we want to decrease the number of buckets in the A-memory, we must increase the number of processors correspondingly. For example, if we allow only 500 buckets in the A-memory of a processor, we would need 100 or more processors in the PP. Thus, there is a memory-processor tradeoff.

In Figure 7c, we plot the B-memory size as a function of the number of processors for various values of N (the number of buckets in the A-memory) and Z_{bs} (the read-out time of the B-memory). As expected, the required size of the B-memory reduces as the number of processors being utilized increases. We note that the figure has eight curves, and that the four curves corresponding to the case where the number of buckets in the A-memory is 10,000 are to the right and the four curves corresponding to the case where the number of buckets in the A-memory is 100,000 are to the left. Thus, if the number of buckets in the A-memory is fixed at 10,000, there must be 16 or more processors in the PP. Similarly, if the number of buckets in the A-memory is fixed at 100,000, then there must be 4 or more processors in the PP. Once again, we notice the inverse dependence between the number of processors and the number of buckets for the A-memory.

Figures 7a, 7b and 7c were all drawn using the information shown in Table I. This table calculates the size of the B-memory for various values of n (the number of processors), N (the number of buckets) and Z_{bs} (the read-out time of a record from the B-memory). $\bar{S}_B$ and $\bar{Q}_B$ represent the average size of the B-memory as calculated by the two different approxima-

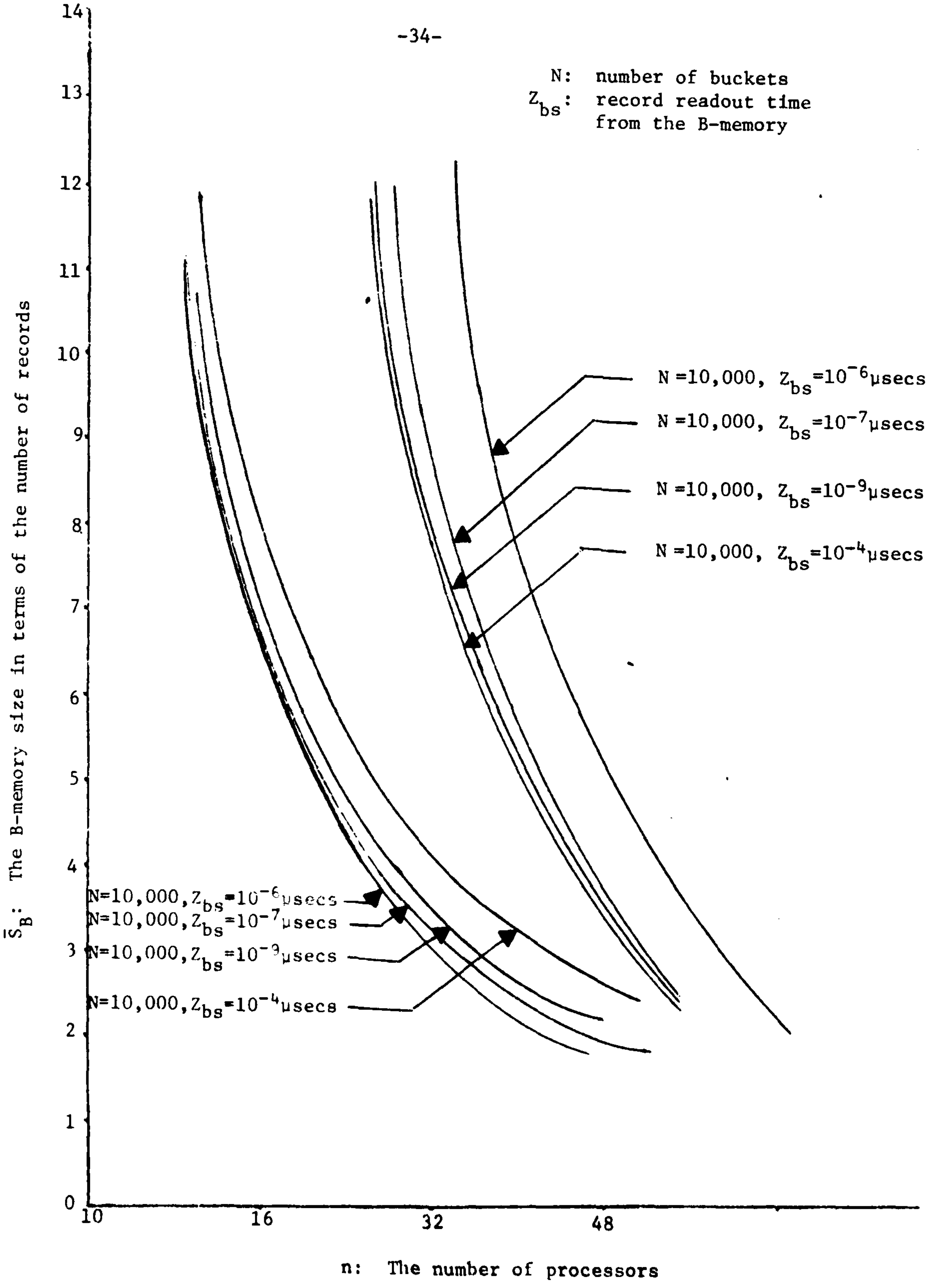

N: number of buckets
Z_{bs}: record readout time from the B-memory
N =10,000, Z_{bs}=10^{-6}μsecs
N =10,000, Z_{bs}=10^{-7}μsecs
N =10,000, Z_{bs}=10^{-9}μsecs
N =10,000, Z_{bs}=10^{-4}μsecs
N=10,000,Z_{bs}=10^{-6}μsecs
N=10,000,Z_{bs}=10^{-7}μsecs
N=10,000,Z_{bs}=10^{-9}μsecs
N=10,000,Z_{bs}=10^{-4}μsecs
$\bar{S}_B$: The B-memory size in terms of the number of records
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
10
16
32
48
n: The number of processors

Figure 7c

For a = 1, Z_{as} = 10 μsecs and Z_{ar} = 55 μsecs

N	n	Z_{bs}	$\bar{S}_B$	S_B95	$\bar{Q}_B$	Q_B95
10,000	32	10^{-9}	1.402	7.672	0.917	6.848
10,000	32	10^{-8}	1.408	7.706	0.922	6.878
10,000	32	10^{-7}	1.475	8.071	0.986	7.247
10,000	32	10^{-6}	2.897	15.855	2.382	15.074
10,000	48	10^{-9}	0.618	3.381	0.221	2.541
10,000	48	10^{-8}	0.620	3.390	0.222	2.551
10,000	48	10^{-7}	0.637	3.486	0.236	2.652
10,000	48	10^{-6}	0.905	4.950	0.466	4.163
100,000	16	10^{-9}	1.169	6.395	0.644	5.244
100,000	16	10^{-8}	1.171	6.407	0.646	5.256
100,000	16	10^{-7}	1.192	6.521	0.665	5.373
100,000	16	10^{-6}	1.463	8.007	0.925	6.894
100,000	32	10^{-9}	0.579	3.169	0.175	2.201
100,000	32	10^{-8}	0.580	3.174	0.175	2.207
100,000	32	10^{-7}	0.589	3.225	0.183	2.261
100,000	32	10^{-6}	0.708	3.873	0.278	2.944
100,000	48	10^{-9}	0.386	2.110	0.060	1.191
100,000	48	10^{-8}	0.386	2.113	0.061	1.194
100,000	48	10^{-7}	0.392	2.146	0.064	1.229
100,000	48	10^{-6}	0.468	2.560	0.110	1.672

TABLE I

ions (see Section 5.2.2). S_B95 and Q_B95 represent the size of the B-memory (as calculated by the same two approximations) in order to be 95% certain that no overflows will occur. We note that there is very good correspondence between the results of the two approximations.

Up to this point, all our observations about the B-memory size were made with the assumption of certain characteristics of the A-memory (e.g., the time to access a block and then read a record is 55 μsecs and the time to read the next record sequentially is 10 μsecs). We now wish to investigate how the size of the B-memory will be affected if we change the characteristics of the A-memory. In order to do this, we calculated the B-memory size for various combinations of the A-memory speed characteristics. The results of the study were interesting. It is discovered that the key parameter with respect to the A-memory (i.e., the one which decides the size of B-memory) was neither one of the two (speed) characteristics themselves, but rather the <u>difference</u> of the two characteristics.

More formally, if Z_{as} is the time to read the next record sequentially from the A-memory, and Z_{ar} is the time to access a block of the A-memory and then read a record from it and $\bar{S}_B$ is the average size of the B-memory, then $\bar{S}_B$ is dependent on $(Z_{ar} - Z_{as})$. We see from our definitions of Z_{ar} and Z_{as}, that $(Z_{ar} - Z_{as})$ is really the time required to access a block of the A-memory called the <u>block access time</u>.

The first observation we make as a result of our experiments is that an A-memory with block access time greater than 45 μsecs is not fast enough for our purposes (that is, the B-memory size approaches infinity). Thus, an A-memory with the block access and record read-out time being 60 μsecs, and record read-out time being 10 μsecs is not good enough for our purposes because the block access time, as the difference, is 50 μsecs. However, an A-memory with the block access and record read-out time being 100 μsecs and the record read-out time being 60 μsecs is good enough for our purposes. In fact, the results of the B-memory sizes obtained for this particular combination of the A-memory speed characteristics is tabulated in Table II.

In Figure 8a (the data used in drawing this figure and Figure 8b is shown in Tables III, IV, V and VI), we plot the B-memory size versus the record read-out time of the A-memory. We note that these curves have fairly small slopes indicating that a change in the record read-out time

Let the following characteristics be fixed

a = 100% (the fraction of target records that participate)

Z_{ar} = 100 μsecs (the block access and record read-out time from the A-memory)

Z_{as} = 60 μsecs (next record read-out time from the A-memory)

We have

N	n	Z_{bs}	$\bar{S}_B$	$S_B 95$	$\bar{Q}_B$	$Q_B 95$
100,000	16	10^{-9}	0.918	5.025	0.413	3.830
100,000	16	10^{-8}	0.919	5.030	0.414	3.836
100,000	16	10^{-7}	0.930	5.088	0.424	3.897
100,000	16	10^{-6}	1.056	5.777	0.540	4.619
100,000	32	10^{-9}	0.393	2.153	0.052	1.095
100,000	32	10^{-8}	0.394	2.155	0.052	1.097
100,000	32	10^{-7}	0.397	2.171	0.054	1.115
100,000	32	10^{-6}	0.431	2.357	0.073	1.323
100,000	48	10^{-9}	0.254	1.390	0.009	0.429
100,000	48	10^{-8}	0.254	1.391	0.009	0.430
100,000	48	10^{-7}	0.256	1.400	0.009	0.439
100,000	48	10^{-6}	0.275	1.502	0.014	0.545

TABLE II

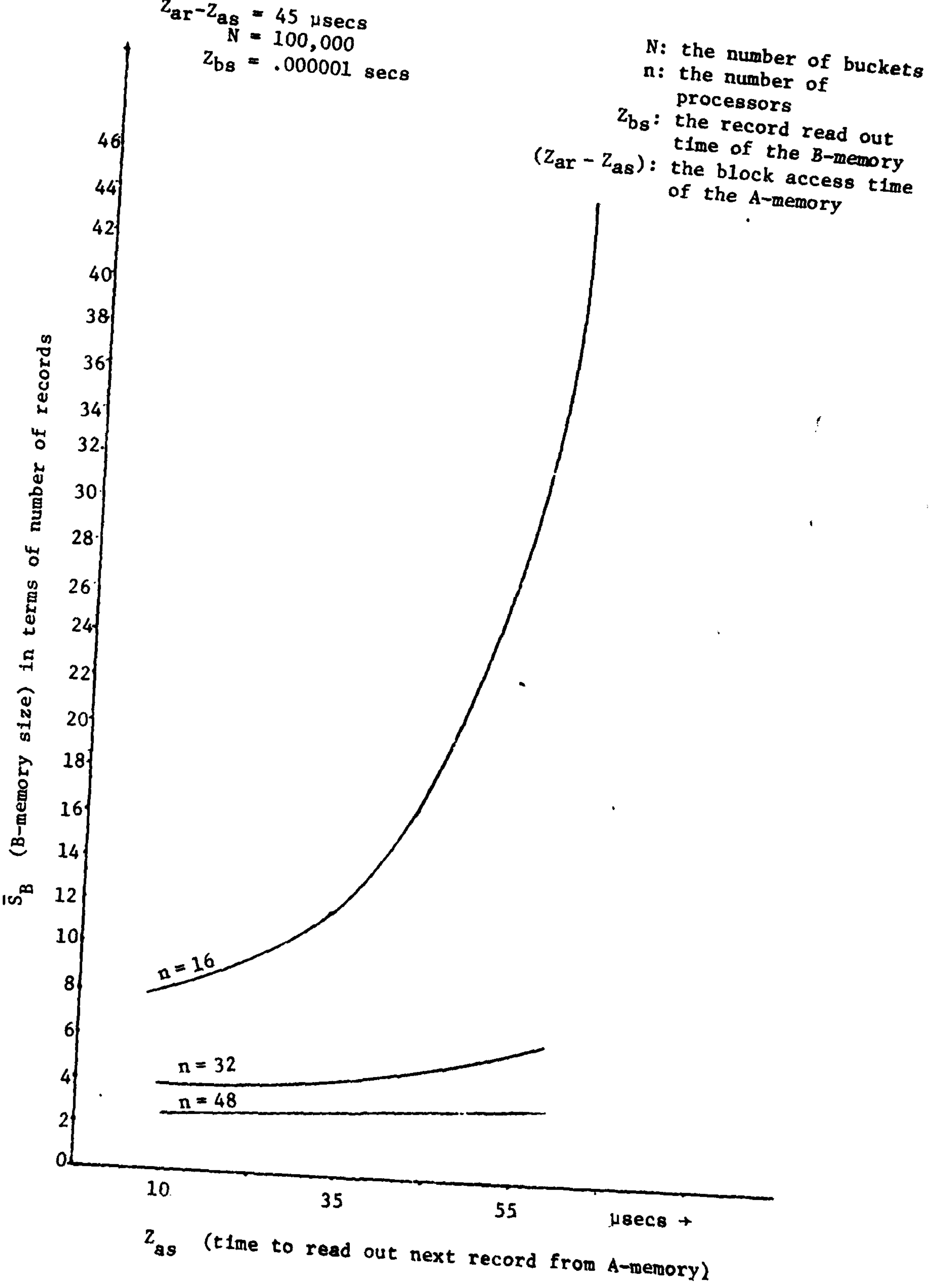

Z_{ar}-Z_{as} = 45 μsecs
N = 100,000
Z_{bs} = .000001 secs
N: the number of buckets
n: the number of processors
Z_{bs}: the record read out time of the B-memory
(Z_{ar} - Z_{as}): the block access time of the A-memory
$\bar{S}_B$ (B-memory size) in terms of number of records
0
2
4
6
8
10
12
14
16
18
20
22
24
26
28
30
32
34
36
38
40
42
44
46
n = 16
n = 32
n = 48
10
35
55
μsecs →
Z_{as} (time to read out next record from A-memory)

Figure 8a

$a = 1$

$Z_{as} = 35$ μsecs

$Z_{ar} = 80$ μsecs

N	n	Z_{bs}	$\bar{S}_B$	S_B95	$\bar{Q}_B$	Q_B95
100,000	16	10^{-9}	1.753	9.590	1.207	8.507
100,000	16	10^{-8}	1.758	9.618	1.212	8.535
100,000	16	10^{-7}	1.810	9.907	1.264	8.827
100,000	16	10^{-6}	2.629	14.385	2.069	13.340
100,000	32	10^{-9}	0.680	3.720	0.255	2.785
100,000	32	10^{-8}	0.681	3.727	0.256	2.793
100,000	32	10^{-7}	0.695	3.802	0.267	2.871
100,000	32	10^{-6}	0.879	4.811	0.427	3.918
100,000	48	10^{-9}	0.426	2.331	0.083	1.428
100,000	48	10^{-8}	0.427	2.335	0.084	1.432
100,000	48	10^{-7}	0.434	2.377	0.088	1.476
100,000	48	10^{-6}	0.534	2.924	0.156	2.057

TABLE III

$a = 1$

$Z_{as} = 55$ μsecs

$Z_{ar} = 100$ μsecs

N	n	Z_{bs}	$\bar{S}_B$	S_B95	$\bar{Q}_B$	Q_B95
100,000	16	10^{-9}	3.110	17.019	2.548	15.993
100,000	16	10^{-8}	3.127	17.112	2.565	16.087
100,000	16	10^{-7}	3.310	18.112	2.746	17.090
100,000	16	10^{-6}	8.357	45.732	7.780	44.744
100,000	32	10^{-9}	0.801	4.383	0.358	3.478
100,000	32	10^{-8}	0.803	4.394	0.360	3.489
100,000	32	10^{-7}	0.823	4.503	0.377	3.601
100,000	32	10^{-6}	1.109	6.070	0.638	5.209
100,000	48	10^{-9}	0.468	2.563	0.110	1.676
100,000	48	10^{-8}	0.469	2.568	0.111	1.681
100,000	48	10^{-7}	0.479	2.620	0.117	1.737
100,000	48	10^{-6}	0.608	3.326	0.212	2.481

TABLE IV

$a = 1$

$Z_{as} = 10$ μsecs

$Z_{ar} = 45$ μsecs

N	n	Z_{bs}	$\bar{S}_B$	S_B95	$\bar{Q}_B$	Q_B95
10,000	16	10^{-9}	0.777	4.253	0.290	3.025
10,000	16	10^{-8}	0.778	4.256	0.291	3.029
10,000	16	10^{-7}	0.783	4.287	0.296	3.063
10,000	16	10^{-6}	0.848	4.639	0.353	3.444
10,000	32	10^{-9}	0.329	1.801	0.021	0.683
10,000	32	10^{-8}	0.329	1.802	0.021	0.684
10,000	32	10^{-7}	0.330	1.808	0.022	0.692
10,000	32	10^{-6}	0.344	1.883	0.028	0.783
10,000	32	10^{-5}	1.401	7.668	0.916	6.839
10,000	48	10^{-9}	0.214	1.171	0.002	0.208
10,000	48	10^{-8}	0.214	1.171	0.002	0.208
10,000	48	10^{-7}	0.215	1.174	0.002	0.211
10,000	48	10^{-6}	0.221	1.211	0.003	0.248
10,000	48	10^{-5}	0.618	3.380	0.220	2.540
100,000	16	10^{-9}	0.516	2.822	0.071	1.300
100,000	16	10^{-8}	0.516	2.823	0.071	1.301
100,000	16	10^{-7}	0.517	2.831	0.072	1.313
100,000	16	10^{-6}	0.533	2.916	0.084	1.432
100,000	16	10^{-5}	1.168	6.394	0.644	5.243
100,000	32	10^{-9}	0.284	1.556	0.007	0.381
100,000	32	10^{-8}	0.284	1.556	0.007	0.381
100,000	32	10^{-7}	0.285	1.559	0.007	0.385
100,000	32	10^{-6}	0.291	1.593	0.009	0.433
100,000	32	10^{-5}	0.579	3.168	0.175	2.201
100,000	48	10^{-9}	0.196	1.075	0.001	0.121
100,000	48	10^{-8}	0.197	1.075	0.001	0.122
100,000	48	10^{-7}	0.197	1.077	0.001	0.124
100,000	48	10^{-6}	0.201	1.097	0.001	0.144
100,000	48	10^{-5}	0.386	2.110	0.060	1.190

TABLE V

$a = 1$

$Z_{ar} = 50$ μsecs

$Z_{as} = 10$ μsecs

N	n	Z_{bs}	$\bar{S}_B$	S_B95	$\bar{Q}_B$	Q_B95
10,000	16	10^{-9}	1.534	8.395	1.002	7.334
10,000	16	10^{-8}	1.537	8.413	1.005	7.353
10,000	16	10^{-7}	1.571	8.595	1.037	7.538
10,000	16	10^{-6}	2.026	11.085	1.482	10.058
10,000	32	10^{-9}	0.462	2.527	0.093	1.515
10,000	32	10^{-8}	0.462	2.529	0.093	1.517
10,000	32	10^{-7}	0.467	2.554	0.096	1.544
10,000	32	10^{-6}	0.519	2.843	0.132	1.860
10,000	48	10^{-9}	0.279	1.525	0.015	0.569
10,000	48	10^{-8}	0.279	1.527	0.015	0.571
10,000	48	10^{-7}	0.281	1.538	0.016	0.583
10,000	48	10^{-6}	0.306	1.674	0.024	0.726
100,000	16	10^{-9}	0.653	3.573	0.179	2.235
100,000	16	10^{-8}	0.653	3.575	0.180	2.238
100,000	16	10^{-7}	0.657	3.598	0.183	2.264
100,000	16	10^{-6}	0.704	3.854	0.222	2.553
100,000	32	10^{-9}	0.344	1.883	0.028	0.787
100,000	32	10^{-8}	0.344	1.884	0.028	0.788
100,000	32	10^{-7}	0.346	1.894	0.029	0.800
100,000	32	10^{-6}	0.368	2.012	0.039	0.935
100,000	48	10^{-9}	0.234	1.280	0.005	0.320
100,000	48	10^{-8}	0.234	1.281	0.005	0.321
100,000	48	10^{-7}	0.235	1.288	0.005	0.327
100,000	48	10^{-6}	0.249	1.363	0.008	0.403

TABLE VI

of the A-memory has only a small impact on the B-memory size. In Figure 8b, we plot the B-memory size versus the block access time of the A-memory. The very large slopes of these curves indicate the very high correlation between the block access time of the A-memory and the size of the B-memory. These curves also indicate to us that the block access time may not exceed 45 μsecs. To summarize the discussion in this paragraph, we will prefer an A-memory with a slow record read-out time and a fast block access time to one with a fast record read-out time and a slow block access time. Intuitively, the reason for this may be explained as follows. During the second phase of the join operation, each target record is hashed to a block of the A-memory. The service time for that record includes an access of that block followed by read-out of records in that block. The number of records in a block can be reduced as much as we want by the use of more processors in the PP and more buckets in the A-memories. Thus, the dominating factor in the service time of a target record becomes the block access time. A slow block access time will cause a large service time, and hence the B-memory as the buffer will soon overflow. This is the reason for the observed phenomenon.

We have seen, in the previous paragraphs, four different ways to reduce the size of the B-memory of a processor. They are to:

(1) increase the number of processors of the ?P,

(2) increase the number of buckets in the A-memory,

(3) decrease the record read-out time of the B-memory, and

(4) decrease the block access time of the A-memory.

Even though each of these recommendations will decrease the size of the B-memory of a processor, the overall cost of the PP may still increase.

In the next section, we present an integrated study of the A-memory, the B-memory and the number of processors in order to arrive at the optimum combination of processors and memories to be placed on a single silicon chip.

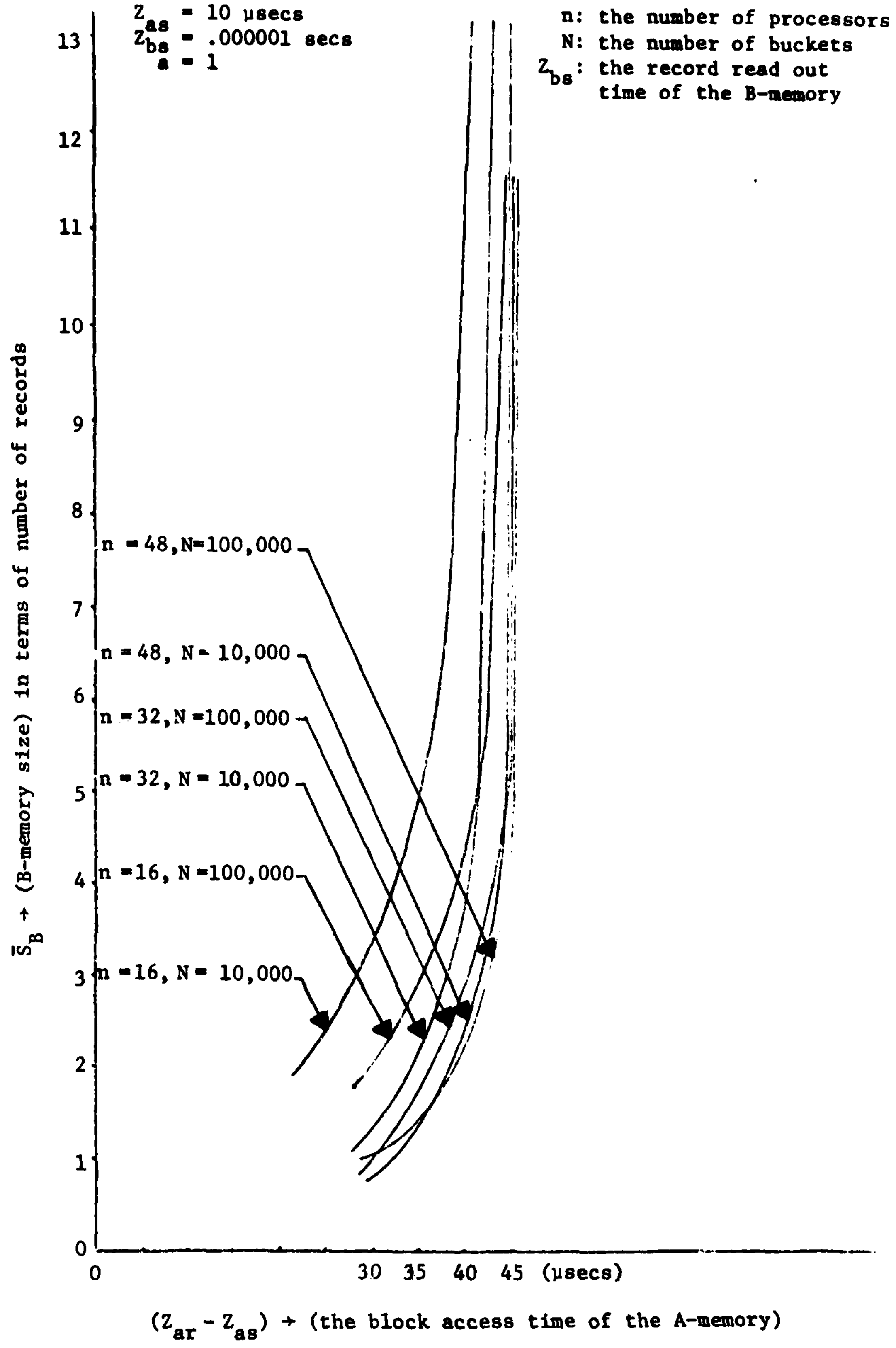
5.4 Design of an Optimal Chip for Use in PP

Our interest in join algorithms using parallel processors was motivated by the following two factors.

(1) The decreasing cost of logic (i.e., cheaper processors).

and

(2) The increasing level of integration possible on a silicon chip



Z_{as} = 10 μsecs
Z_{bs} = .000001 secs
a = 1
n: the number of processors
N: the number of buckets
Z_{bs}: the record read out time of the B-memory
n = 48, N=100,000
n = 48, N = 10,000
n = 32, N =100,000
n = 32, N = 10,000
n = 16, N =100,000
n = 16, N = 10,000
13
12
11
10
9
8
7
6
5
4
3
2
1
0
0
30
35
40
45
(μsecs)
$\bar{S}_B$ → (B-memory size) in terms of number of records
$(Z_{ar} - Z_{as})$ → (the block access time of the A-memory)

Figure 8b

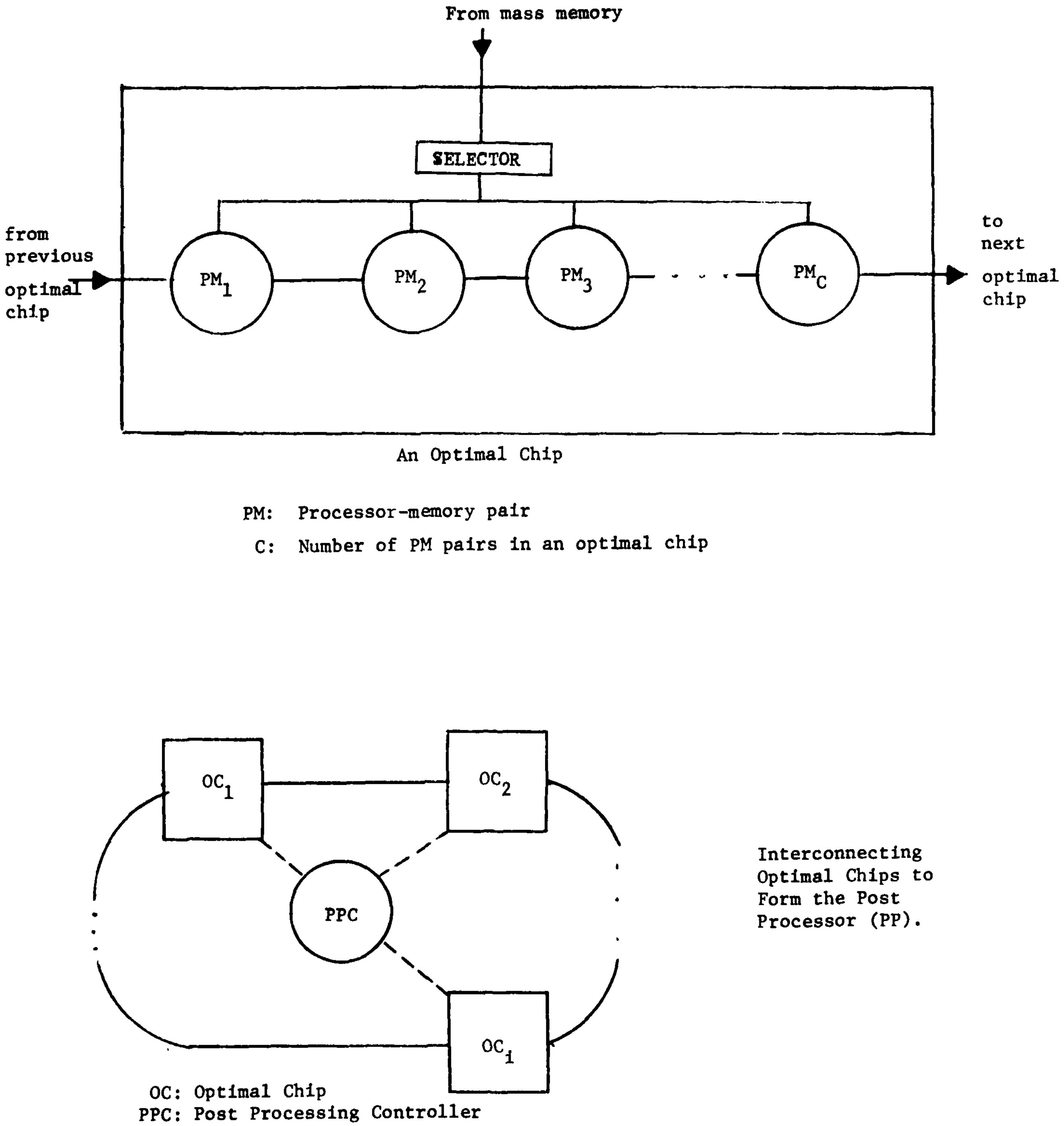
(i.e., VLSI).

Thus, our description of the parallel join algorithm would not be complete without a discussion on how the post processor used to perform the join may actually be implemented using large-scale integrated technology. In the previous sections, we developed equations that could be used to calculate how many processors, how much A-memory, and how much B-memory must be used in order to achieve a certain join speed. In this section, we will estimate the 'optimal' combination of processors and memories that may be placed on a chip, where a particular combination is considered optimal if it provides the best performance for a certain fixed cost. A chip which contains this optimal combination of processors and memories will be called an optimal chip.

We will consider the problem of finding an optimal chip for two different time frames. First, we will design a chip which would be optimal given the present state of the art of large scale integrated technology. Next, we will design an optimal chip for the year 1990, basing our calculations on the projected increases in the level of integration (number of transistors and amount of memories that can be placed on a chip with acceptable yield) possible by that year. We do the latter only because of our belief that the optimal chip for 1990 would nct be a simple extension of the optimal chip for 1980. Rather, we believe that it will be radically different. Beyond 1990, however, radical changes in the design of the optimal chip are unlikely. The reasons for this will become clearer as we go along.

Having designed such an optimal chip, PP may then use a single chip to attain a certain join speed, or it may use two interconnected chips to attain approximately twice that speed, or it may use three interconnected chips to attain approximately thrice that speed, and so on. Owing to the simplicity and hardware extensib'lity of our interconnection scheme among processors (see Figure 2), the optimal chips may also be easily interconnected in the desired fashion to obtain faster speeds.

In Figure 9, we show an optimal chip, and also the interconnection scheme between such chips. Each chip is connected to two other chips, and all the chips are connected together in a circular fashion. Each chip consists of a number of processor-memory (PM) pairs. The number of PM pairs to be placed on a chip, and the amount and type (RAM, CCD, etc) of



An Optimal Chip

PM: Processor-memory pair

C: Number of PM pairs in an optimal chip

Interconnecting Optimal Chips to Form the Post Processor (PP).

OC: Optimal Chip
PPC: Post Processing Controller

Figure 9. Description and Interconnection of Optimal Chips

memory that is attached to each processor in a PM pair, will be calculated in the following sections. An important point to note about the optimal chip is that it needs only three pins. Thus, the optimal chip is not pin-limited. In other words, the number of processor-memory pairs that can be placed on a chip is not limited by the number of pins but only by the level of integration.

A. Designing an Optimal Chip for Today

First, we note that the present state of the art allows for the fabrication of 70,000 transistors on a single chip [23]. Thus, 70K bits of random-access memory (RAM) can be placed on a single chip. Next, we are interested in knowing how many charge-coupled devices (CCD) can be placed on a chip, since the A-memories of the PP are made of the CCD memory. The maximum density CCD chip available today is 256K bits. Earlier, we had stated that 70K transistors can be put on a chip. Thus, we may roughly equate 1 bit of the CCD memory to $\frac{70}{256}$ ($\approx$0.25) of a transistor. This rough equivalence allows us to discuss in terms of either transistors or CCDs as units per chip in the sequel.

Finally, we need to estimate the number of transistors needed to make a processor. In [23], it is estimated that about 40,000 transistors are required to make a general-purpose processor. However, the processors we propose to use in the PP are very simple processors and they are special-purpose rather than general-purpose processors. In fact, these processors only need to compare two records on the basis of certain special attribute values in them, and concatenate the two records if the values in the two records are equal. In addition, the processors are required to calculate a simple hash function. Without consulting electrical engineers, we can only estimate how much simpler than a general-purpose processor the processor of the PP is. For the purposes of the ensuing calculations, we approximate that the processor of PP is only $\frac{1}{8}$th as complex as a general-purpose processor and thus requires only $\frac{1}{8}$th as many transistors. Then, each processor in the PP will need 5000 transistors.

We assume, as before, that the maximum possible size of the source set is 100,000 records, each record being of size 500 bytes. In order to store these many source records, at least 4 x 10^8 bits of the A-memory (made of CCDs) are required in the PP. Even if as many as 1,000 processors are employed in the PP, we find that each of these 1000 processors will need $\frac{4 \times 10^8}{10^3} = 4 \times 10^5$ bits of the CCD memory. Present day levels of integration will not allow us to put this amount of memory along with a processor on a chip. Thus, our first design decision regarding an optimal chip is that the A-memory will not be integrated into the chip containing the processors, but will be on separate chips.

The B-memory (which is made of RAM), on the other hand, may certainly be integrated into a chip along with processors, since it is usually of very small size (a few K bytes). In Table VII, we show various combinations of processors and B-memory sizes that can be placed on a chip using present-day levels of integration. [These calculations are made using the fact that 70K transistors can be put on a chip and that one processor in the PP requires 5K transistors]. For example, we may choose to integrate a single processor and 65K of the B-memory, or we may choose to put two processors, each with 30K of the B-memory together on a chip, and so on. We shall label these allowable (by fabrication technology) combinations of processors and B-memory sizes as Type I, Type II, ..., Type VII. Our purpose is to determine which of these seven chip types would be optimal for use in the PP.

The first point to be noted is that the use of more processors and memories on a chip decreases the need for transferring records across chip boundaries, from one chip to another. This is a big advantage, since transfers that are local to a chip are an order of magnitude faster and consume less power than record transfers across chip boundaries [23]. From this viewpoint, chip type VII is the best, since it has the greatest number of processors and memories (7) on a single chip.

Let us now take a different approach to the choice of an optimal chip type. Since, ultimately, many optimal chips will be interconnected as in Figure 9 to form a PP, we would like to compare the performance obtained by connecting together a large number of chips of Type I with that obtained by connecting together a large number of chips of Type II, and so on. For illustrative purposes, let us consider that the PP is built using 200 chips and that it must be able to service records at the rate of at least one every 50 μsecs (which is the maximum read out rate of a parallel transfer disk as calculated in Section 5.3). Consider that the chip with one processor and 65K of the B-memory (Type I chip) is used to construct the PP.

Chip Type	Number of processors	Amount of the B-memory (K-bits)	Amount of the B-memory per processor (K-bits)
I	1	65	65.0
II	2	60	30·0
III	3	55	18·33
IV	4	50	12·5
V	5	45	9·0
VI	6	40	6·67
VII	7	35	7·0

TABLE VII: Various Combinations of Processors and Memories that can be put on a Chip with Present-day Technology

Then, it may be shown by use of the equation derived in Appendix 3 (for S_B95, the size of B-memory) that the A-memory (constructed on different chips) attached to each processor must contain at least 1,065 buckets. In other words, the use of fewer than 1,065 buckets of the A-memory will require the size of the B-memory (buffer) attached to a processor to be greater than 65K bits, thus making use of chip Type I infeasible. Knowing the number of buckets in each A-memory allows us to calculate (using the equation derived previously in Section 5.2.1), the size of each A-memory. It may be easily shown that the A-memory attached to a processor must be at least of size 44.6 M bits, and that the total size of the A-memory required is 8,920 M bits. Finally, the equation for $\bar{x}$ or E(service time), derived in Appendix 2, may be used to show that the record service time is 49.71 μsecs. This explains the first row in Table VIII. Similar calculations are made for each of the other six chip types and the results for all these chip types is presented in Table VIII.

We reiterate that the results in Table VIII are made assuming that 200 chips are used in PP. We wish to compare the PP created using 200 chips of Type I (in terms of speed and cost) with the PP created using 200 chips of Type II, and so on. Our first observation is that 200 chips of Type I will cost the same as 200 chips of Type II, or Type III, or any of the other types, because each of the chips are of similar complexity (since they all need 70,000 transistors). We note, however, that chip Type I has the lowest total A-memory requirement and also the slowest record service time. Chip Type VII, on the other hand, has the largest A-memory requirement and also the fastest record service time. In order to compare the different chip types, we need to either make the A-memory requirement uniform and compare the different service times, or make the service time uniform and compare the A-memory requirement. We choose to do the former in keeping with our definition of optimal as one giving the best performance for a fixed cost.

The normalized results are shown in Table IX. It is noticed that making the total A-memory requirement uniform across all chip types has caused all service times to be also uniform across chip types. Thus, all chip types are equally attractive since they have the same cost and record service time. However, even though the service times per record are the same for all chip types, a PP which uses more processors will have to service fewer records. Thus, the total time of join (= number of records

Chip Combination Type	Number of Buckets per A-memory	Size of the A-memory per processor (M bits)	Total size of the A-memory (M bits)	Record Service Time (μsecs)
I	1065	44.6	8,920	49.71
II	550	23.0	9,200	49.56
III	381	15.91	9.546	49.38
IV	298	12.42	9,936	49.20
V	250	10.04	10,040	49.01
VI	220	9.133	10,960	48.80
VII	199	8.246	11,544	48.60

TABLE VIII. Performance of a PP Using Different Chip Types

Chip Combination Type	Number of Buckets Per A-memory	Size of A-memory Per Processor (M bits)	Total size of all A-memory (M bits)	Record Service Time (μsecs)
I	1400	58.0	11600	48.60
II	700	29.0	11600	48.60
III	467	19.33	11600	48.60
IV	350	14.50	11600	48.60
V	280	11.60	11600	48.60
VI	233	9.67	11600	48.60
VII	200	8.29	11600	48.60

TABLE IX. Normalizcd Results of PP Using Various Chip Types

serviced x time taken to service a record) is minimized if more processors are used in the PP. Hence, we will prefer the chip type with the largest number of processors in it, and this is chip Type VII. Furthermore, the use of chip Type VII will minimize record transfers across chip boundaries, thus contributing to a decrease in join time.

We conclude that the optimal chip type is the one with 7 processors and 35K of the B-memory (5K per processor) on a chip. At least 200 such chips will have to be interconnected in order to provide a service rate of one every 50 μsecs. Use of more than 200 chips in PP would mean that the service rate would increase to more than one every 50 μsecs. Such use of many optimal chips is possible owing to the hardware extensibility of our PP design and simplicity of interconnection scheme.

B. Designing an Optimal Chip for 1990

Using the figures provided in [23], we estimate that, by the year 1990, between 10 and 12 million transistors can be put on a chip. Let us assume, for the purposes of this calculation, that 11 million transistors can be put on a chip in 1990. This means that large amounts of the A-memory (made of CCDs) can now be integrated into a chip along with a processor. This makes the optimal chip of 1990 radically different from the optimal chip of today, since the latter did not incorporate any A-memory. We now need to find out what combinations of processors and (both A and B) memories should be placed on a chip. Obviously, many combinations of processors and memories may feasibly be placed on a chip. Our purpose will be to choose a combination which is optimal in the sense that it gives the best performance for a certain fixed cost, where, by performance, we mean the total time to do a join operation.

As before, the optimal chip is designed so that a large number of them (say 200) may be interconnected to form the PP which should be able to service records at the rate of at least one every 50 μsecs. An examination of the equation for $\bar{\bar{x}}$ or E(service time) derived in Appendix 2 shows us that the service time depends upon the product nN, where n is the number of processors in PP and N is the number of buckets in the A-memory. If we choose the maximum size of the source set (C_{smax}) to be 100,000 records, the record size to be 500 bytes, the number of records in a block (R) to be 10, the B-memory record read-out time to be 10^{-8} seconds, the

A-memory block access time to be 45 μsecs, and the A-memory record read-out time to be 10 μsecs, we see that nN must be 2 x 10^5 for the record service time to be the desired value of 50 μsecs. Now, the total size of A-memory (see Section 5.2.1) is

$$\simeq nx\left(\frac{C_{smax}}{n} + NR\right)$$

$$= C_{smax} + nNR \qquad \text{records}$$

We see that the size of the A-memory depends only on the product nN (C_{smax} and R are constants) which is fixed at 2 x 10^5 in order to obtain a record service time of 50 μsecs. Thus, the total size of the A-memory is fixed, whatever combination of processors and memories goes on a chip, and is given by (10^5 + 2 x 10^5 x 10) records. This means that each chip will contain 42 M bits of the A-memory. Since 1 bit of the A-memory is equivalent to 0.25 transistors, each chip will utilize 10.5 million (out of 11 million) transistors for the A-memory. The remaining 0.5 million transistors on a chip must be divided up be*ween processors and the B-memory.

In Table X, we show the combinations of processors and the B-memory that add up to 0.5 million transistors. We need to decide first, for each combination shown, if it is feasible or not. That is, for the given A-memory size (fixed earlier) and the given number of processors, is the buffer size (B-memory size) large enough to handle an arrival rate of one record every 50 μsecs. If the B-memory is large enough, the combination is feasible and; otherwise, it is infeasible. The equation derived in Appendix 3 was used to check each of the listed combinations for feasibility. Only the first 30 combinations of Table X were found to be feasible.

It is now concluded that the combination with the largest number of processors on a chip is the optimal one. The reasons are similar to the reasons used earlier. First, the total cost of 200 chips of any combination type is the same since each chip combination is of the same complexity (in terms of number of transistors). Second, although the service times per record are the same for all chip types, a PP which uses more processors will have to service fewer records. Thus, the total time of join is minimized if more processors are used in the PP and this will happen if there are more processors on a chip. Moreover, use of a combination with the largest number of processors on a chip will minimize record transfers across chip bound-

Number of Processors	Total size of B-memory (K-bits)	Amount of B-memory per Processor (K-bits)	
1	495.00	495.00	
2	490.00	245.00	
3	485.00	161.67	
4	480.00	120.00	
5	475.00	95.00	
6	470.00	78.33	
7	465.00	66.43	
8	460.00	57.50	
9	455.00	50.56	
10	450.00	45.00	
11	445.00	40.45	
12	440.00	36.67	
13	435.00	33.46	
14	430.00	30.71	
15	425.00	28.33	
16	420.00	26.25	
17	415.00	24.41	
18	410.00	22.78	
19	405.00	21.32	
20	400.00	20.00	
21	395.00	18.81	feasible
22	390.00	17.73	
23	385.00	16.74	
24	380.00	15.83	
25	375.00	15.00	
26	370.00	14.23	
27	365.00	13.52	
28	360.00	12.86	
29	355.00	12.24	
30	350.00	11.67	
31	345.00	11.13	infeasible
32	340.00	10.63	
33	335.00	10.15	
34	330.00	9.71	
35	325.00	9.29	
36	320.00	8.89	
37	315.00	8.51	
38	310.00	8.16	
39	305.00	7.82	
40	300.00	7.50	
41	295.00	7.20	
42	290.00	6.90	
43	285.00	6.63	
44	280.00	6.36	
45	275.00	6.11	
46	270.00	5.87	
47	265.00	5.64	
48	260.00	5.42	
49	255.00	5.20	
50	250.00	5.00	

TABLE X. Various Combinations of Processors and Memories that can be Placed on a Single Chip in 1990.

Number of Processors	Total Size of B-memory (K-bits)	Amount of B-memory per Processor (K-bits)
51	245.00	4.80
52	240.00	4.62
53	235.00	4.43
54	230.00	4.26
55	225.00	4.09
56	220.00	3.93
57	215.00	3.77
58	210.00	3.62
59	205.00	3.47
60	200.00	3.33
61	195.00	3.20
62	190.00	3.06
63	185.00	2.94
64	180.00	2.81
65	175.00	2.69
66	170.00	2.58
67	165.00	2.46
68	160.00	2.35
69	155.00	2.25
70	150.00	2.14
71	145.00	2.04
72	140.00	1.94
73	135.00	1.85
74	130.00	1.76
75	125.00	1.67
76	120.00	1.58
77	115.00	1.49
78	110.00	1.41
79	105.00	1.33
80	100.00	1.25
81	95.00	1.17
82	90.00	1.10
83	85.00	1.02
84	80.00	0.95
85	75.00	0.88
86	70.00	0.81
87	65.00	0.75
88	60.00	0.68
89	55.00	0.62
90	50.00	0.56
91	45.00	0.49
92	40.00	0.43
93	35.00	0.38
94	30.00	0.32
95	25.00	0.26
96	20.00	0.21
97	15.00	0.15
98	10.00	0.10
99	5.00	0.05
100	0.00	0.00

TABLE X. (cont.)

aries thus contributing to a decrease in join time.

We conclude that the optimal chip type is the one with 30 processors, 42 M bits of the A-memory (1.4 M bits per processor) and 350K bits of B-memory (11.67 K bits per processor) on a chip. At least 200 such chips will have to be interconnected in order to provide a service rate of one every 50 μsecs. Use of fewer than 200 such chips in the PP will result in a service rate lower than one record every 50 μsecs. Similarly, use of more than 200 such chips in PP will result in a service rate higher than one record every 50 μsecs.

6. INEQUALITY AND M-WAY JOIN OPERATIONS

In this section, we will describe how the method used for joining record sets may be easily extended to handle inequality joins and m-way joins.

6.1 Requirements for and Phases of the Inequality Join

As in the case of the equality join, an inequality join is an operation that involves two record sets, namely, the source set and the target set and two attributes, i.e., the source attribute of the source set and the target attribute of the target set. Furthermore, it is necessary that the values of the source and target attributes be drawn from the same domain.

There are four possible types of inequality joins, depending upon which of the four inequality operators (i.e., $<$, $\leq$, $>$, $\geq$) is involved in the join. Let us denote an inequality join between record sets A and B as A 'op' B, where op is one of the four inequality operators. Furthermore, let us call the set of records produced as a result of the join operation as the result set. Then, each record in the result set C of the inequality join A 'op' B is a concatenation of two records r_1 and r_2 where r_1 is a record in the source set A, r_2 is a record in the target set B, and op holds between the source attribute value in r_1 and the target attribute value in r_2. In Figure 10, we show an example of a greater-than join.

In order to handle inequality joins, it is necessary that the hashing function HASH in use satisfies the following criterion. Given two attribute values x and y such that $x < y$, we require that $HASH(x) \leq HASH(y)$. Such a hashing function can be easily found. For example, suppose we are hashing salaries, p, to blocks of the A-memory. Let the A-memory have N blocks. We then want the hashing function to return a block number between one and N. Furthermore, let the highest possible salary be MAXSAL. Then, a hashing function that satisfies our requirements would be

$$HASH(p) = \left\lceil \frac{p}{MAXSAL} \right\rceil N$$

Let us now describe the method used to do inequality joins. As in the case of equality join, the operation proceeds in two phases. In the first phase, the source records are read into the B-memories and then hashed and stored in the blocks of the A-memories. Next, the target records are read

A: The source set of the following five records

(<Name, HSIAO>, <Salary, 2000>)

(<Name, JAI>, <Salary, 1000>)

(<Name, KERR>, <Salary, 2000>)

(<Name, JOHN>, <Salary, 5000>)

(<Name, JACOB>, <Salary, 3000>)

B: The target set of the following three records

(<Benefit, 2000>, <Dept, TOY>)

(<Benefit, 3000>, <Dept, SALES>)

(<Benefit, 4000>, <Dept, FINANCE>)

Let the source attribute be Salary and the target attribute be Benefit. Then the result set where Salary of A is greater than Benefit of B consists of the following four records:

(<Name, JOHN>, <Salary, 5000>, <Benefit, 2000>, <Dept, TOY>)

(<Name, JOHN>, <Salary, 5000>, <Benefit, 3000>, <Dept, SALES>)

(<Name, JOHN>, <Salary, 5000>, <Benefit, 4000>, <Dept, FINANCE>)

(<Name, JACOB>, <Salary, 3000>, <Benefit, 2000>, <Dept, TOY>)

Figure 10. An example of a greater-than join

into the B-memories. Each processor now does the following for each target record in its B-memory. It reads the record from the B-memory and hashes the target attribute value of the record to a block (say, i) of the A-memory. Now, the processor accesses and searches records in every block from the first block to the i-th block (in the case of a less-than or less-than-or-equal-to join) or from the i-th block to the N-th block (in the case of a greater-than or greater-than-or-equal-to join). The source attribute values of these records are now examined in order to determine if they will participate in the join. The join is performed between the target record and all the qualified source records, and the concatenated records are placed in the C-memory. We note that we do not make use of associative memories for the inequality join.

In summary, we note that the essential difference between the method for equality joins and the method for inequality joins occurs in the second phase. For an equality join, each target attribute value will be hashed to a block of the A-memory and all source records in that block and that block alone will be examined for possible participation in the join operation. For a less-than or less-than-or-equal-to (greater-than or greater-than-or-equal-to) join, each target attribute value will be hashed to a block of the A-memory and all source records in that block and in all other blocks with lower (higher) block numbers will be examined for possible participation in the join operation. Consequently, an inequality join is a slower operation in comparison with the equality join.

6.2 Operations of the M-Way Join

Until this time, we have considered only the manner in which two record sets are joined. However, we may be required first to join two record sets and then to join the result set with another record set. We call this operation a 3-way join. Similarly, the m-way join of A_1, A_2, ..., A_m requires that the sets A_1 and A_2 be first joined to produce an intermediate result set, say B_1. The set B_1 must now be joined with A_3 to produce another intermediate result set B_2. This process continues until B_{m-2} is joined with A_m to produce the final result set B_{m-1}. In Figure 11, we show an example of a 3-way join between 3 record sets, namely, the NAME record set, the EMP record set and the DEPT record set. The NAME and EMP record sets are first joined using Emp# as both the source and target attributes.

The NAME record set consists of the following two records.	The EMP record set consists of the following three records.	The DEPT record set consists of the following three records.
(<Name, HSIAO>, <Emp# ,1>)	(<Emp#, 1>, <Dept#, 2>)	(<Dept#, 2>, <No. of Employees, 20>)
(<Name, JAI>, <Emp#, 2>)	(<Emp#, 2>, <Dept#, 3>)	(<Dept#, 3>, <No. of Employees, 25>)
	(<Emp#, 3>, <Dept#, 4>)	(<Dept#, 4>, <No. of Employees, 19>)

(1) Join NAME and EMP; Use Emp# as source and target attributes.

(2) Join the result set of (1) and DEPT; Use Dept# as source and target attributes.

The result set of (2) is as follows:

(<Name, HSIAO>, <Dept#, 2>, <No. of Employees, 20>)

(<Name, JAI>, <Dept#, 3>, <No. of Employees, 25>)

Figure 11. An example of a 3-way Join of NAME, EMP and DEPT

The resulting set is now joined with the record set DEPT using Dept# as both the source and target attributes. We see that a 3-way join requires 2 joins to be performed and 2 source and target attributes to be specified. Similarly, an m-way join requires (m-1) joins to be pei.formed and requires the specification of (m-1) source and target attributes. Let us refer to these source and target attributes as SA_1, SA_2, ..., SA_{m-1} and TA_1, TA_2, ..., TA_{m-1}, respectively. In the following paragraphs, we shall describe the method employed to do the m-way join in the PP.

Let the result set of an m-way join of A_1, A_2, ..., A_m be represented as A_1 x A_2 x ... x A_m. First, the set A_1 is read and stored in the A-memories. Next, the set A_2 is read in the B-memories and the intermediate set A_1 x A_2 is formed and stored in the C-memories. Next, the set A_3 is read in the B-memories and the intermediate set A_1 x A_2 x A_3 is formed and stored in the A-memories. The process continues in this way, with the intermediate sets being stored alternately in the A and C-memories. The intermediate set A_1 x A_2 x ... x A_i will be stored in the C-memories if i is even and in the A-memories if i is odd. The final result set A_1 x A_2 x ... x A_m will be formed and stored in the C-memories if i is even and in the A-memories if i is odd.

In order to derive some time measures, let us make the assumption that each of the record sets A_1, A_2, ... A_m is clustered and stored in a separate content-addressable cylinder of the mass memory (MM). This implies that each set can be read in one disk revolution of MM due to its tracks-in-parallel-readout capability (see Section 2). Let us now estimate the number of revolutions needed to perform the m-way join and produce the set A_1 x A_2 x ... x A_m. In the first revolution, the set A_1 is read in and stored in the A-memories. In the second revolution, the set A_2 is read in, joined with A_1, and the result set A_1 x A_2 is stored in the C-memories. In an earlier section, we had indicated how to adjust the B-memory (buffer memory) sizes in order to ensure that joins are performed in the PP at the disk transfer rate. Thus, each additional revolution of the mass memory will cause an additional set A_i to be read in and an additional intermediate result set A_1 x A_2 x ... x A_i to be formed and stored in either the A or C-memories depending on whether i is odd or even. Finally, at the end of m revolutions, the result set A_1 x A_2 x ... x A_m would have been created. Thus, an m-way join will take m revolutions of the mass memory. In general, if we assume that each record set requires c revolutions of the mass memory to read it out, the join operation takes mc revolutions. That is, the join operation is performed as fast as the participating record sets are being read out. To put it another way, perfect pipelining can be achieved between the mass memory and the post processor during the execution of the join operation. Of course, such perfect piplining will be achieved only if the A and C-memories are large enough to hold all the source records and also to hold each intermediate set created during the join operation in its entirety. Otherwise, we may require multiple passes over some of the record sets which are then required to be retrieved from the mass memory more than once.

7. COMPARISON WITH OTHER METHODS

In this section, we intend to make a survey of some of the other available join methods and to make a comparison between these other methods and our own methods. We compare the methods based upon whether or not they can support natural and implicit joins, inequality joins and m-way joins. The methods are also compared in terms of the time complexity of the join, where the time complexity is given in terms of the cardinalities of the source, target and result sets, and also in terms of the number of revolutions of the secondary (mass) storage device needed to effect the join. Finally, the methods are compared on whether join processing may be overlapped with the input of the source and target sets. We begin with a brief description of each of the various methods. The comparison between these methods is presented in the form of a table at the end of this section.

The machine known as the content-addressable file store (CAFS) [17] supports only the implicit join operation and does not support the natural join. The method uses a single-bit wide random-access store in order to aid the join operation. The operation proceeds essentially as follows. First, each source attribute value is read and a bit in the random-access store is marked. The address of the bit to be marked is determined either by hashing the source value to an address, or by using a pre-compiled coupling index [17]. The target records are now read. Each target attribute value will cause a bit in the random-access store to be examined. As before, the address of the bit to be examined is determined either by hashing the target value to an address, or by using a pre-compiled coupling index. If the examined bit has been marked, then this target record (from which the target value was derived) will participate in the join and is output. Notice that the process involves one scan of the source set, one scan of the target set and the creation of the result set. Hence, the complexity of the join is

$$O(C_s + C_t + C_r).$$

In an implicit join operation, it is the case that $C_r \leq C_t$ and hence $O(C_s + C_t + C_r)$ approaches $O(C_s + C_t)$.

There are problems with the hashing scheme and the pre-compiled coupling index scheme. If the method of pre-compiled indices is used, then it is necessary that we know, before hand, which sets will have to be joined.

Additionally, it requires extra space to store the coupling indices and extra time to update these indices during an update. On the other hand, if a hashing scheme is used, there is always the possibility of errors in the result set. This is caused by the collisions due to hashing. In DBC we only use hashing to locate blocks and do not use hashing for locating records. This is because we do not discard the join values since we use these values to identify the participating records. Thus, DBC produces no error in the result set.

It seems very likely that the CAFS method can overlap input/output (I/O) with join processing, since the join algorithm is such that it may begin with the arrival of the first source record. By this we mean that the join operation will take as many revolutions (of the mass storage device to complete) as the number of revolutions needed to read out the source and target sets. However, in order to support m-way joins with I/O overlap, it will require to use an additional random-access bit store. Also, if the method of hashing is used, the number of errors in the result set will go up when m-way joins are performed. That is, a 3-way join will result in more errors than a 2-way join, a 4-way join will result in more errors than a 3-way join, and so on. If the hashing function used has the special property suggested in Section 6.1, then inequality joins may be supported.

The method used in the context addressed segmented sequential memory (CASSM) for doing joins [18] is very similar to the above method. Thus, most of the comments we made above about the CAFS method will be relevant to the CASSM method. There are two main differences, however, between these methods. First, CASSM uses a physical random-access bit store for each track of the secondary storage, and so the implementation of one virtual random-access bit store, addressable by all cells (a cell is a track of a disk and its associated logic), requires the address of a bit to be passed from one cell to the next until it arrives at the appropriate cell. This may require additional rotational delay so that join processing may require more revolutions of the secondary store than required to read the source and target sets. Second, while CAFS uses a separate unit for doing joins which is removed from the secondary storage device itself, CASSM does the join directly on the tracks of the disk which constitute the secondary store. Thus, comments regarding whether the join operation can be overlapped with I/O are not relevant since no separate unit is used to do the join. This also implies that the intermediate sets created during the execution of the m-way join must be stored in the secondary storage itself.

The relational associative processor (RAP) [19] also supports only an implicit join operation. The operation may be understood in terms of Figure 12, which is reproduced from [19]. The source set is stored in x cells, where a cell consists of some circulating memory and associated logic. Similarly, the target set is stored in y cells. The circulating memories have the start-stop feature, so that the time for one rotation of the circulating memory can be varied and depends upon how much processing may have to be done by the logic on the records stored in the circulating memory. The entire operation is controlled by the RAP controller and proceeds as follows. The first cell containing the source values of the first source set is read and buffered at the RAP controller, which consists of z blocks. Then, a block of source values is loaded into the cells containing the target set, and these cells are initiated for processing. This block loading of source values is repeated until all of the buffered source values are processed; then the next cell of source values are buffered at the RAP controller and the above operations are repeated until all cells containing source values are processed.

Since the cardinality of the source set is C_s, and the records of the source set are stored across x cells, each source set cell has $\frac{C_s}{x}$ records. Similarly, each target set cell has $\frac{C_t}{y}$ target records. Since the buffer has z blocks, each block can hold $\frac{C_s}{xz}$ source values. Thus, $\frac{C_s}{xz}$ source values are processed in a batch. Each of the $\frac{C_t}{y}$ target records in a target cell have to be compared against these $\frac{C_s}{xz}$ source values, and this takes $\frac{C_t}{y} \cdot \frac{C_s}{xz}$ time if each cell logic has one processor, or $\frac{1}{n} \frac{C_t}{y} \frac{C_s}{xz}$ time if each cell logic has n processors. The above is the time to process a single block. Since xz blocks have to be processed in all, the total join time is

$$O(\frac{C_s C_t}{ny}).$$

Thus, the time complexity of join in RAP is of $O(C_s C_t)$.

In terms of number of revolutions, the join operation takes xz revolutions, where x is typically about 15 or 20, and z is typically between one and three. However, the number of revolutions is not a good time com-

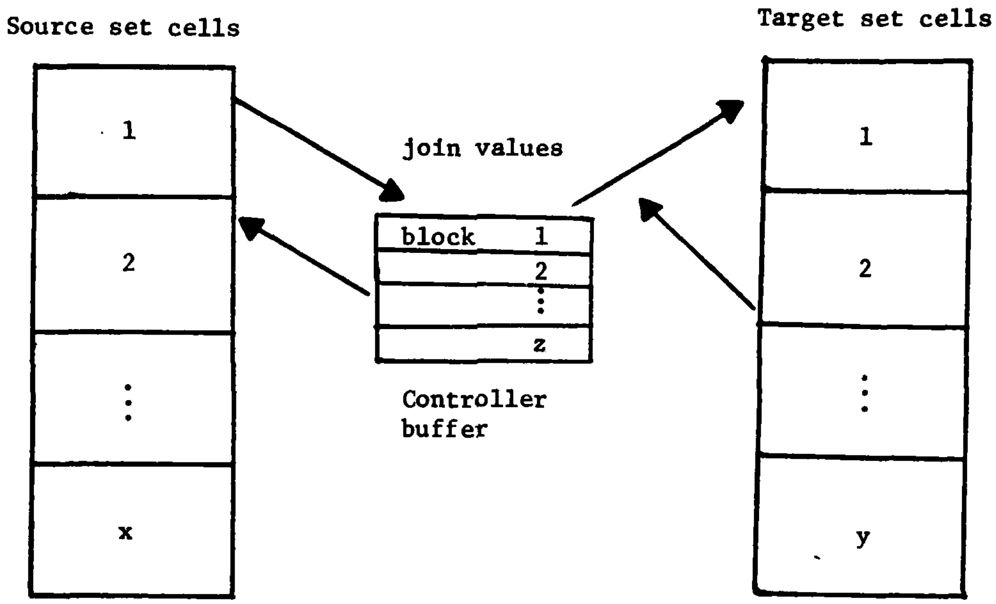


Figure 12. The RAP join method

plexity measure for the RAP join method since the revolution time of the circulating memory, as has been already pointed out, is variable.

RAP is capable of doing inequality joins. However, in order to be able to do m-way joins, it must create and store temporary sets in the circulating memories. It is not clear how this may be done, but there is no reason to suppose that it cannot.

Up to this point, we discussed machines that could only do implicit joins. The remaining machines that we discuss will be capable of doing both implicit and natural joins.

The relational machine of Shaw [13] is configured as a hierarchy of associative storage devices. At the top of this hierarchy is the primary associative memory (PAM), which is a fast content-addressable memory containing between 10K and 1M bytes. The secondary associative memory (SAM) is where the database is stored and may consist of parallel head-per-track disks as in the CASSM and older RAP designs, or modified moving-head disks as in the DBC design. The source and target sets are retrieved from SAM and placed in PAM, where the actual join takes place. Since the PAM is quite small, the source and target sets are brought to the PAM in pages, and this requires many accesses to the source and target sets on the SAM (i.e., many revolutions of the secondary storage devices). In [13], it is shown that the join operation in the PAM is of complexity

$$O(3C_s + C_r).$$

The factor C_t is not a part of the time complexity because the author did not include the time taken to input the target set into the PAM. In keeping with our policy of including the input time in complexity calculations, the time complexity is of order

$$O(3C_s + C_r + C_t).$$

Thus, the complexity is of the same order of magnitude as the complexity of the DBC method, even though costlier hardware such as PAM has been utilized to achieve it.

In terms of number of secondary storage revolutions, it is shown in [13] that at least

$$(\frac{C_s}{P} + \frac{C_t}{P})(1 + w)$$

revolutions will be needed to complete the join, where P is the size of the

PAM in terms of number of records, and w is a 'waste factor' and is typically a small fraction. Typically, $\frac{C_s}{P}$ and $\frac{C_t}{P}$ range from ten to twenty. Thus, a typical figure for the number of secondary storage revolutions needed to complete the join is in the range from twenty to forty.

A detailed look at the join algorithm proposed in [13] reveals that the source and target sets have to be present, in their entirety, before the actual join may begin. Thus, it is not possible to overlap the join processing with I/O of the source and target sets as in the DBC. Also, m-way joins cannot be done without incurring waits, unless a second PAM is used. Finally, the method cannot be utilized to do inequality joins.

Let us now consider the method used in a machine called DIALOG [20] to do natural joins. The source set is initially stored in a buffer B_1. The source values from the source set are extracted and stored in an associative sequential memory. Target records are read out from the secondary store and pass through an associative processor which decides if the target records will participate in the join (by comparing the target attribute values with the source attribute values stored in the associative sequential memory). Target records that will participate in the join are allowed to pass through and be stored in a buffer B_2. The join processor then joins target records in B_2 with source reco-ds in B_1 (using a buffer B_3 to decide which records in B_1 should be joined with a particular record in B_2) and outputs them. It is easily seen that the join is of complexity

$$O(C_s + C_t + C_r).$$

Clearly, Buffer B_1 must be large enough to hold the entire source set. If not, multiple passes over the source and target sets will be necessary and this may entail up to

$$\frac{C_s}{\text{size of } B_1} \text{ x } \text{(number of revolutions to retrieve target set)}$$

revolutions of the secondary store for completion of the join operation. Thus, the choice is between a very large (and expensive) random-access buffer and a very large number of revolutions to do the join (typically 100).

The method can be used to do inequality joins. However, in order to do m-way joins without incurring waits, it is necessary to have an additional buffer (which will operate like B_1) and an additional associative sequential memory to hold source values.

We now consider the scheme used in DIRECT [21]. The join operation is performed using a set of query processors and a set of CCD page frames. In this system, the query processors and CCD page frames are connected to each other using a cross-bar switch, so that all processors can access all page frames. We agree with the sentiments expressed in [19], that the cross-bar switch of DIRECT (though much simpler than the conventional cross-bar switch) may not be cost-effective and may reduce the performance of a full-scale system.

Let us assume that the target set resides in y pages. Also, let us assume that n processors are used for the join operation. Each processor will join $\frac{y}{n}$ pages of the target set with the entire source set. Thus, each processor joins $\frac{C_t}{n}$ target records with C_s source records and produces approximately $\frac{C_r}{n}$ result records. Thus, the join operation has complexity

$$O(\frac{C_t C_s}{n} + \frac{C_r}{n}).$$

The nice thing about the method is that n processors can do the join n times faster than a single processor can, and, hence, the method is very efficient. Its drawback is the fact that the time complexity is of the order of the product of the source and target set cardinalities.

The DIRECT method is the only one, besides our own, which addresses the issue of using more than one processor to speed up the join. That is, these are the only two methods which employ parallel join algorithms for doing natural joins. We feel that the study of multiple processor algorithms is important owing to the falling cost of processing logic. However, the DIRECT scheme uses a more complicated and less extensible interconnection scheme for processors and memories than we do.

An important point to note is that the secondary store associated with DIRECT does not have any selection capability. Thus, the source and target sets have to be retrieved, in their entirety, into the CCD page frames, and the selection of the parts of these sets that will participate in the join has to be done after the retrieval from secondary store. Hence, unlike in the DBC, join operation cannot be overlapped with I/O, since join can only begin after I/O followed by selection.

This method can be utilized to do inequality joins and m-way joins. However, the temporary sets created during the execution of an m-way join operation may be created on half-pages which must be 'compressed' [22] to full pages before the next step may be executed in an m-way join operation. This page compression operation which is necessitated owing to the page fragmentation problem will slow down the m-way join operation.

Some additional problems with the DIRECT join method are as follows. First, unlike the DBC post processor which is an SIMD machine, the DIRECT is an MIMD machine. Thus, there is the need for a more sophisticated controller. Unlike the post processing controller of DBC which only needs to issue simple broadcast messages to the processors, the DIRECT controller needs to keep track of which processors are executing what queries, decide on the optimal number of processors to execute a join, allocate temporary page frames for intermediate results created during the execution of the join operation, enforce concurrency control, and so on. Second, the DIRECT system treats the results of basic relational algebra operations as temporary sets and stores these sets in temporary page frames. However, since DIRECT is an MIMD machine, it is quite likely that the execution of another query of higher priority may cause the temporary set created to be paged out to secondary storage before it is used in an m-way join operation. Thus, the m-way join operation may require several accesses to secondary storage to complete.

Having considered each of the other methods in turn, it is now time to consider the DBC join method. First, it is of complexity $O(\frac{C_s}{n} + C_t + \frac{C_r}{n})$, where n is the number of processors. This complexity is as good as, or better than, that of any other known join method. Together with DIRECT, they are the only methods, which use multiple processors for doing natural joins. Unlike DIRECT, however, the interconnection scheme among the processors is very simple, regular and extensible. This is an important consideration for VLSI implementation of the post processor.

The DBC join method is capable of doing inequality joins and m-way joins and of overlapping I/O with join processing. It is the only method that can accomplish an m-way natural join in m revolutions typically.

In Figure 13, we summarize our discussion in the form of a comparison chart. Each method is characterized on the basis of the absence or presence of various qualities (like I/O overlap, ability to do inequality and m-way joins, etc.). They are also compared on the basis of time complexity in terms of set cardinalities and in terms of number of secondary store revolutions.

(1) Implicit Join

(2) Natural Join

(3) Time Complexity in Terms of Set Cardinalities

(4) Time Complexity in Numbers of Secondary Store Revolutions (Typical)

(5) M-way Joins without Rotational Delays

(6) Inequality Joins

(7) Overlapped I/O

(8) Possibility of Errors

(9) Use of Unit Different from Storage

(10) No. of Processors in Additional Unit

	(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)	(9)	(10)
CAFS	✓	x	$O(C_s + C_t + C_r)$	2	A.H.	A.C.	A.H.	✓	✓	1
CASSM	✓	x	$O(C_s + C_t + C_r)$	> 2	A.C.	A.C.	N.A.	x	x	N.A.
RAP	✓	x	$O(C_s C_t)$	15-60	A.C.	✓	N.A.	x	x	N.A.
SHAW	✓	✓	$O(C_s + C_t + C_r)$	20-40	A.H.	x	x	x	✓	1
DIALOG	✓	✓	$O(C_s + C_t + C_r)$	100	A.H.	✓	x	x	✓	1
DIRECT	✓	✓	$O(C_s C_t + C_r)$	?	A.C.	✓	x	x	✓	1 or more (MIMD)
DBC	✓	✓	$O(C_s + C_t + C_r)$	2	✓	✓	✓	x	✓	1 or more (SIMD)

Explanations:

✓ - Yes ; x - No ; N.A. - Not Applicable; A.H. - Additional Hardware Needed;
A.C. - Additional Change in algorithm; ? - Unknown or hard to estimate;
C_s - Cardinality of source set; C_t - Cardinality of target set;
C_r - Cardinality of result set.

Figure 13. Comparison Chart of Various Join Methods

8. CONCLUSIONS

This paper repeats in detail the algorithm proposed in [4] to perform relational equality joins in a database computer known as the DBC. Two improvements to the proposed method are made herein: The first improvement is due to the addition of a new type of memory for each processor, called the C-memory. The C-memory is used to store the results of the join operation. It is also a vital component during the m-way join operation. The second improvement involves the replacement of a single and fast associative memory (AM) by several smaller and slow associative memories (ams). Two advantages accrue as a result of this replacement. First, we show that each of the ams needs to operate at a slower rate than the single AM. This implies that we may use random-access memory (RAM) to realize the ams. The second advantage is an improvement in the time complexity of the join operation. It is shown that the time complexity of the join is linear in the cardinalities of the source, target and result sets. We postulate that no join algorithm can have better time complexity than this one.

The paper also provides a thorough queueing analysis of the process of join as it is carried out in a specialized component of DBC known as the post processor (PP). In doing the queueing analysis, we ignore the presence of associative memories, since the associative memory merely speeds up the algorithm. Hence, the throughput achievable by the PP without the use of associative memories can be used as the lower bound of the PP performance measure.

PP consists of a number of interconnected processors, each of which is associated with three sets of memories. The queueing analysis shows that if PP of DBC is to perform joins at a rate commensurate with the output rate of records (i.e., relational tuples) from the mass memory (MM) of DBC, then the memories associated with PP must satisfy certain constraints with respect to speed and size. In this respect, some constraints are more crucial than others. The paper clearly indicates the order of importance of the various constraints.

The paper also indicates how to select an optimum chip design for PP, where the design is considered optimal if it gives the best performance for a certain fixed cost. The method is one that may be employed by DBC systems designers in order to arrive at the optimal values for

(1) the number of processors on a chip, and

(2) the size of the memories to be placed on a chip.

Finally, the paper describes the extensions of the DBC method. The extensions enable PP to perform relational inequality joins and m-way joins.

We conclude this paper with a favorable comparison of the DBC join method with the join schemes proposed on some other database machines.

REFERENCES

[1] Codd, E.F., "A Relational Model of Data for Large Shared Data Banks," CACM, Vol. 13, No. 6, June 1970.

[2] Kannan, K., "The Design of a Mass Memory for a Database Computer," Proceedings of the Fi^th Annual Symposium on Computer Architecture, April 1978, Palo Alto, California, pp. 44-50; Also available in Hsiao, D.K. and Kannan, K., "The Architecture of a Database Computer -- Part III: The Design of the Mass Memory and its Related Processors," Technical Report, OSU-CISRC-TR-76-3, The Ohio State University, Columbus, Ohio, Dec. 76.

[3] Menon, M.J., and Hsiao, D.K., "The Access Control Mechanism of a Database Computer (DBC)," Fifth Annual Workshop on Computer Architecture for Non-numeric Processing, Pacific Grove, California, March 11-19, 1980; Also available in Banerjee, J., Hsiao, D.K., and Menon, M.J., "The Clustering and Security Mechanisms of a Database Computer (DBC)," Technical Report, OSU-CISRC-TR-79-2, The Ohio State University, Columbus, Ohio, April 1979.

[4] Hsiao, D.K., and Menon, M.J., "The Post Processing Functions of a Database Computer," Technical Report, OSU-CISRC-TR-79-6, The Ohio State University, Columbus, Ohio, July 1979.

[5] Hsiao, D.K., and Menon, M.J., "Parallel Record Sorting Methods for Hardware Realization," Technical Report, OSU-CISRC-TR-80-7, The Ohio State University, Columbus, Ohio, July 1980.

[6] Hsiao, D.K., Kannan, K., and Kerr, D.S., "Structure Memory Designs for a Database Computer," Proceedings of ACM 77 Conference, Oct. 1977, Seattle, Washington; Also available in Kannan, K., Hsiao, D.K., and Kerr, D.S., "A Microprogrammed Keyword Transformation Unit for a Database Computer," Proceedings of the Tenth Annual Workshop on Microprogramming, (Oct. 1977), Niagara Falls, New York.

[7] Kleinrock, L., "Queueing Systems I and II," John Wiley, New York, 1975.

[8] Denning, P.J., and Buzen, J.P., "The Operational Analysis of Queueing Network Models," Computing Surveys, Vol. 10, No. 3, September 1978, pp. 225-261.

[9] Sevcik, K.C., Levy, A.I., Tripathi, S.K., and Zahorjan, J.L., "Improving Approximations of Aggregated Queueing Network Subsystems," Computer Performance, K.M. Chandy and M. Reiser (Eds.), Elsevier North-Holland Inc., New York, 1977, pp. 1-22.

[10] Gaver, D.P., "Diffusion Approximations and Models for Certain Congestion Problems," Journal of Appl. Probab., Vol. 5, 1968, pp. 607-623.

[11] Ampex Corp., PTD - 930x Parallel Transfer Drive, Product Description 3308829-01, October 1978.

[12] IBM S/370-360, IMSL Library 1, Vol. II, Edition 6, 1977.

[13] Shaw, D., "A Relational Database Machine Architecture," Proceedings of the Fifth Annual Workshop on Computer Architecture for Non-numeric Processing, Pacific Grove, California, March 11-19, 1980.

[14] Banerjee, J. and Hsiao, D.K., "Performance Evaluation of a Database Computer in Supporting Relational Databases," Proceedings of the Fourth International Conference on Very Large Data Bases, Berlin, Federal Republic of Germany, September 1978, pp. 319-329.

[15] Banerjee, J. and Hsiao, D.K., "A Methodology for Supporting Existing CODASYL Databases with New Database Machines," Proceedings of ACM'78 Conference.

[16] Banerjee, J., Hsiao, D.K., and Ng, F., "Database Transformation, Query Translation and Performance Analysis of a Database Computer in Supporting Hierarchial Database Management," IEEE Transactions on Software Engineering, March 1980.

[17] Babb, E. "Implementing a Relational Database by Means of Specialized Hardware," ACM Transactions on Database Systems, Vol. 4, No. 1, March 1979, pp. 1-29.

[18] Copeland, C.P., Lipovski, G.J. and Su, S.Y.W., "The Architecture of CASSM: A Cellular System for Non-Numeric Processing," Proceedings of the First Annual Symposium on Computer Architecture, Dec. 1973, pp. 121-128.

[19] Oflazer, K., and Ozkarahan, E.A., "RAP.3 - A Multi-Microprocessor Cell Architecture for the RAP Database Machine," Proceedings of the International Workshop on High-Level Language Computer Architecture, May 1980, Fort Lauderdale, Florida.

[20] Wah, B.W., and Yao, B.S., "DIALOG - A Distributed Processor Organization for Database Machine," Proceedings of the National Computer Conference, 1980, pp. 243-253.

[21] DeWitt, D.J., "DIRECT - A Multiprocessor Organization for Supporting Relational Database Management Systems," Proceedings of the Fifth Annual Symposium on Computer Architecture, April 1978, pp. 182-189.

[22] DeWitt, D.J., "Query Execution in DIRECT," Proceedings of the ACM-SIGMOD 1979 International Conference of Management of Data, May 1979, pp. 13-22.

[23] Patterson, D.A. and Sequin, C.H., "Design Considerations for Single-Chip Computers of the Future," IEEE Transactions on Computers, Vol. C-29, No. 2, pp. 108-109, February 1980.

APPENDIX 1: THE INTER-ARRIVAL TIME AND SERVICE TIME DISTRIBUTIONS IN THE SINGLE PROCESSOR CASE

We repeat the definitions of the various parameters which will be needed in our derivations as follows:

N : Number of buckets in the A-memory.

C_s : Cardinality of the source set.

C_t : Cardinality of the target set.

v : Number of distinct values of the join attribute.

y : Time taken by a processor to join two records.

Z_{ar} : Time taken to access a given block and to read (write) a record from (to) the block of the A-memory.

Z_{as} : Time taken to read (write) next record from (to) the A-memory.

Z_{bs} : Time taken to read (write) next record from (to) the B-memory.

p : The probability that a source or target record is selected from the mass memory for the join.

r : Inter-arrival time of records to PP at maximum read-out rate of MM, i.e., parallel read-out disks.

R : Number of records that will fit in a block.

$P(b)$: The Probability that a bucket will contain b records. For the uniprocessor case, assuming that we hcve an equal probability of assigning a record to any of the available buckets, the number of records b assigned to a bucket will be distributed binomially as

$$P(b) = \binom{pC_s}{b} \left(\frac{1}{N}\right)^b \left(1 - \frac{1}{N}\right)^{(pC_s - b)}$$

We are interested in cases where both pC_s and N are large, and where the average number of records assigned to a bucket $\frac{pC_s}{N}$ is nearly equal to the bucket size R. The binomial distribution is then very close to the Poisson distribution with parameter $\frac{pC_s}{N}$. Thus,

$$P(b) = \frac{e^{\frac{-pC_s}{N}} \left(\frac{pC_s}{N}\right)^b}{b!}$$

with,

$$E(b) = \frac{pC_s}{N}$$

and,

$$E(b^2) = \frac{2pC_s}{N}$$

P(b,j,k) : The probability that a bucket of b records will contain exactly j records with a particular join value k. In this case, since the b records are selected from among a set of pC_s records having v distinct values for the join attribute,

$$P(b,j,k) = \frac{\binom{\frac{pC_s}{v}}{j}\binom{\frac{(v-1)pC_s}{v}}{b-j}}{\binom{pC_s}{b}}$$

The assumption made, of course, is that there are likely to be as many records having one particular join value as there are records having another join value.

We are now ready to analyze the inter-arrival and service distributions.

Inter-arrival Time Distribution

The records of the target set are read from the MM and arrive directly at the B-memory from the MM after selection. Each of the C_t target records has a probability p of being selected for the join, and hence, of being read into the B-memory. Thus, the probability that i records out of C_t will be selected is binomially distributed as

$$P_2(i) = \binom{C_t}{i} p^i (1-p)^{(C_t-i)}$$

For large C_t (target set cardinality), we approximate this as a Poisson distribution. Thus, the probability that i records out of C_t will be selected is

$$P_2(i) = \frac{e^{-pC_t}(pC_t)^i}{i!}$$

The time taken to read C_t target records and select i of them is rC_t. Thus, the probability of i arrivals in a time interval of rC_t is

$$P_i(rC_t) = P_2(i) = \frac{e^{-pC_t}(pC_t)^i}{i!}$$

Finally, the probability of i arrivals in time t is

$$P_i(t) = \frac{e^{\frac{-pt}{r}}\left(\frac{pt}{r}\right)^i}{i!}$$

Since the arrival process is Poisson, the inter-arrival time is exponentially distributed and is

$$\frac{1}{r}\, e^{\frac{-pt}{r}} \quad .$$

The mean of the distribution is

$$\frac{r}{p} \; .$$

The variance of the distribution is

$$\frac{r^2}{p^2}$$

Service Time Distribution

The service time consists of the time to read a target record from the B-memory, hash its join value (say k) to a block in the A-memory, read all records in that block and all overflow records of that block, and join the record with all source records in the block and overflow area with join value k. We will ignore the hashing time in our calculations. Also, the time to read all records in a block is taken as (1) the time to access the block and to read sequentially all the records of that block, and (2) the time to access overflow blocks and read their overflow records. The actual service time will depend on the architectural details of the processors and memories. Thus, if a Direct Memory Access (DMA) is used to read records from the A-memory and write them to the C-memory, then the time to read records from the A-memory and write them to the C-memory need not be included in the service time. Conversely, if no DMAs are used, then these two times must be added to the service time. In our model, we only include the time to read records from the A-memory and not the time to store records to the C-memory. Thus, service time

$$= Z_{bs} + Z_{ar} + (b-1)Z_{as} + jy \qquad \text{(for } b \leq R\text{)}$$

$$= Z_{bs} + Z_{ar} + (b-R)Z_{ar} + (R-1)Z_{as} + jy \qquad \text{(for } b > R\text{)}$$

where b is the number of records in a bucket and j is the number of records in the bucket with join value k.

We will now proceed to calculate E(service time) or $\bar{x}$, E(service time^2) or $\overline{x^2}$ and σ^2(service time).

$$E(\text{service time}) = \sum_{b=o}^{R} \sum_{j} (Z_{bs} + Z_{ar} - Z_{as} + bZ_{as} + jy)P(b)P(b,j,k)$$

$$+ \sum_{b=R+1}^{\infty} \sum_{j} (Z_{bs} + Z_{ar} - Z_{as} + bZ_{ar} + R(Z_{as} - Z_{ar}) + jy)P(b)P(b,j,k).$$

$$= \sum_{b=o}^{R} Z_{bs}P(b) + (Z_{ar} - Z_{as})P(b) + Z_{as}bP(b) + yP(b) \sum_{j} jP(b,j,k)$$

$$+ \sum_{b=R+1}^{\infty} (Z_{bs} + Z_{ar} - Z_{as})P(b) + Z_{ar}bP(b) + R(Z_{as} - Z_{ar})P(b)$$

$$+ yP(b) \sum_{j} jP(b,j,k).$$

Since $\sum_{j} j\, P(b,j,k) = \sum_{j} j \dfrac{\binom{\frac{pC_s}{v}}{j}\binom{\frac{(v-1)pC_s}{v}}{b-j}}{\binom{pC_s}{b}} = \dfrac{b}{v}$, we have

$$E(\text{service time}) = \sum_{b=o}^{R} (Z_{bs} + Z_{ar} - Z_{as})P(b) + Z_{as}bP(b) + \frac{ybP(b)}{v}$$

$$+ \sum_{b=R+1}^{\infty} (Z_{bs} + Z_{ar} - Z_{as})P(b) + Z_{ar}bP(b) + R(Z_{as} - Z_{ar})P(b) + \frac{ybP(b)}{v}$$

$$= Z_{bs} + Z_{ar} - Z_{as} + \frac{ypC_s}{vN} + \sum_{b=o}^{R} Z_{as}bP(b) + \sum_{b=R+1}^{\infty} Z_{ar}bP(b)$$

$$+ \sum_{b=R+1}^{\infty} R(Z_{as} - Z_{ar})P(b)$$

Let $\sum_{b=s}^{\infty} P(b)$ be represented by the notation Q(s).

Then, $\sum_{b=R+1}^{\infty} bP(b) = \dfrac{pC_sQ(R)}{N}$

and, $\sum_{b=R+1}^{\infty} b^2P(b) = \dfrac{p^2C_s{}^2Q(R)}{N^2}$

So, $E(\text{service time}) = Z_{bs} + Z_{ar} - Z_{as} + \frac{ypC_s}{vN} + \frac{Z_{as}pC_s(1-Q(R+1))}{N}$

$$+ \frac{Z_{ar}pC_sQ(R)}{N} + R(Z_{as} - Z_{ar})Q(R).$$

$$E(\text{service time}^2) = \sum_{b=o}^{R} \sum_{j} (Z_{bs} + Z_{ar} - Z_{as} + bZ_{as} + jy)^2 P(b)P(b,j,k)$$

$$+ \sum_{b=R+1}^{\infty} \sum_{j} (Z_{bs} + Z_{ar} - Z_{as} + bZ_{ar} + R(Z_{as} - Z_{ar}) + jy)^2 P(b)P(b,j,k).$$

Since, $\sum_{j} jP(b,j,k) = \frac{b}{v}$

and, $\sum_{j} j^2(P(b,j,k)) = \sum_{j} j^2 \frac{\binom{\frac{pC_s}{v}}{j}\binom{\frac{(v-1)pC_s}{v}}{b-j}}{\binom{pC_s}{b}} = \frac{b}{v^2(pC_s - 1)}(PC_s(b+p-1) - bp)$

after simplifying. We have, $E(\text{service time}^2)$

$$= Z_{bs}^2 + \frac{2y^2(pC_s - v)p^2C_s{}^2}{v^2(pC_s - 1)N^2} + \frac{y^2p^2C_s{}^2(v-1)}{v^2(pC_s - 1)N} + (Z_{ar} - Z_{as})^2$$

$$+ 2(Z_{ar} - Z_{as})Z_{bs} + \frac{2(Z_{ar} - Z_{as})ypC_s}{vN} + 2(Z_{ar} - Z_{as})(Z_{as} - Z_{ar})RQ(R+1)$$

$$+ \frac{2yZ_{bs}pC_s}{vN} + \frac{Z_{as}^2P^2C_s{}^2(2 - Q(R))}{N^2} + \frac{2Z_{as}Z_{bs}pC_s(1 - Q(R))}{N}$$

$$+ \frac{2(Z_{ar} - Z_{as})Z_{as}pC_s(1 - Q(R))}{N} + \frac{2(Z_{ar} - Z_{as})Z_{ar}pC_sQ(R)}{N}$$

$$+ \frac{2yZ_{as}p^2C_s{}^2(2 - Q(R))}{vN^2} + \frac{Z_{ar}^2p^2C_s{}^2Q(R)}{N^2} + \frac{2Z_{bs}Z_{ar}pC_sQ(R)}{N}$$

$$+ R^2(Z_{as} - Z_{ar})^2Q(R+1) + \frac{2yR(Z_{as} - Z_{ar})pC_sQ(R)}{N} + 2Z_{bs}R(Z_{as} - Z_{ar})Q(R+1)$$

$$+ \frac{2Z_{ar}R(Z_{as} - Z_{ar})pC_sQ(R)}{N} + \frac{2Z_{ar}yp^2C_s{}^2Q(R)}{vN^2}$$

Finally, $\sigma^2(\text{service time}) = E(\text{service time}^2) - (E(\text{service time}))^2$

$$= \frac{2y^2(pC_s - v)p^2C_s{}^2}{v^2(pC_s - 1)N^2} + \frac{y^2p^2C_s{}^2(v-1)}{v^2(pC_s - 1)N} + R^2(Z_{as} - Z_{ar})^2(Q(R+1) - Q^2(R+1))$$

$$+ \frac{Z_{as}^2p^2C_s{}^2(1 + Q(R) - Q^2(R))}{N^2} + \frac{2yZ_{as}p^2C_s{}^2}{vN^2} + \frac{Z_{ar}^2p^2C_s{}^2(Q(R) - Q^2(R))}{N^2}$$

$$+ \frac{2yRpC_s(Z_{as} - Z_{ar})P(R)}{vN} + \frac{2Z_{ar}Z_{as}RpC_s(Q(R) + Q(R + 1) - 2Q(R)Q(R + 1))}{N}$$

$$- \frac{2Z^2_{ar}RpC_sQ(R)(1 - Q(R + 1))}{N} - \frac{2Z_{as}Z_{ar}p^2C_s{}^2(Q(R) - Q^2(R))}{N^2}$$

$$- \frac{2Z^2_{as}pC_sRQ(R + 1)(1 - Q(R))}{N} - \frac{y^2p^2C_s{}^2}{v^2N^2}$$

APPENDIX 2: THE INTER-ARRIVAL TIME AND SERVICE TIME DISTRIBUTIONS IN THE MULTIPROCESSOR CASE

Service Time Distribution

The calculations here are similar to the calculations in Appendix 1. In fact, the final results for E(service time), E(service time^2) and σ^2(service time) may be obtained from the results for these same quantities derived in Appendix 1, by the replacement of all occurrences of N by the product nN (since there are now n processors each of which store $\frac{1}{n}$ of the source records).

Inter-arrival Time Distribution

Figure 4 shows the model used to derive the first and second moments of the inter-arrival distribution. As the figure clearly shows, there are two factors that contribute to the arrival rate of target records to the B-memory of a particular processor. The first is the arrival of records directly from the disks of mass memory. The second factor is the arrival rate of records from the previous processor in the broadcast sequence. We note that each processor discards a fraction (1-q) of the records processed by it. A target record will be discarded by a processor if

(1) The AM indicates that it cannot participate in the join or

(2) The record has already propagated through all the processors.

In this study, we will ignore the presence of the AM.

A. E(inter-arrival time)

In the figure, D is the expected rate of record arrivals from the mass memory and A is the expected rate at which records are processed and output by a processor (this is different from the service rate). By the principle of flow balance [8], at equilibrium, the net arrival rate of target records to a processor must be equal to the net rate at which target records are being output from the processor. Thus,

$$D + q.A = A$$

$$\text{and } A = \frac{D}{1-q}$$

Thus, E (net arrival rate of target records to a processor) = E (rate at which records are output from a processor)

$$= E(A) = \frac{1}{1-q} \; . \; E(D)$$

For the uniprocessor case, we had calculated $E(D) = \frac{p}{r}$. Here, $E(D) = \frac{p}{nr}$ since the output from the mass memory is now streamed to n different processors. Thus,

$$(\text{net arrival rate}) = \frac{1}{(1-q)} \cdot \frac{p}{nr}$$

and

$$E\ (\text{inter-arrival time}) = \frac{(1-q)\cdot nr}{p}$$

B. σ^2(inter-arrival time)

In order to approximate the second moment of the inter-arrival time, we use the approximations of Sevcik, et al. [9]. To simplify the ensuing discussion, we define

$$\propto(x) = \frac{\sigma^2(x)}{(E(x))^2} - 1$$

for any random variable x. Furthermore, let Arrival be a random variale designating the time between two consecutive arrivals, Disk be a random variable designating the time between two consecutive arrivals from the mass memory disk, Department be a random variable designating the time between two consecutive record departures (after service), and Branch be a random variable designating the time between the processing of two consecutive records which are not discarded.

From Figure A,

$$\propto(\text{Branch}) = q \cdot \propto(\text{Departure})\ldots \qquad (1)$$

where q has been defined earlier in this Appendix.

From Figure B,

$$\propto(\text{Departure}) = \delta^2 \propto(\text{service time}) + (1-\delta^2) \propto (\text{Arrival})\ldots \ (2)$$

where,

$$\delta = \lambda\bar{x}$$

and $\lambda = E(\text{arrival rate})$

and $\bar{x} = E(\text{service time})$

Finally, from Figure C,

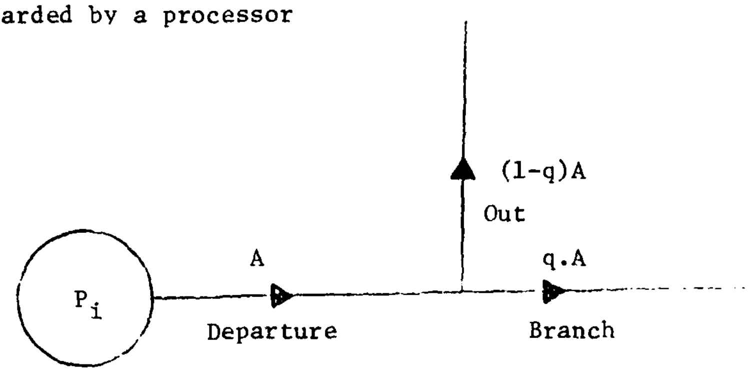
$$\propto(\text{Arrival}) = \left(\frac{E(\text{Arrival})}{E(\text{Disk})}\right)^2 \propto(\text{Disk}) + \left(\frac{E(\text{Arrival})}{E(\text{Branch})}\right)^2 \propto(\text{Branch})$$

$$= (1-q^2)\propto(\text{Disk}) + q^2\propto(\text{Branch})$$

But,

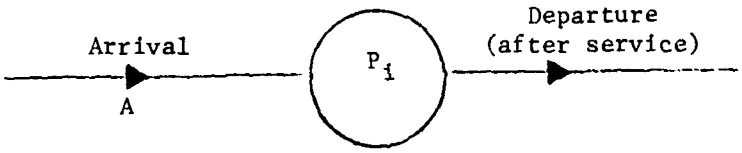
$$\propto(\text{Disk}) = \frac{\sigma^2(\text{Disk})}{(E(\text{Disk}))^2} - 1 = 0$$

D: E(Arrival rate from MM disks)
A: E(net arrival rate) = E(net departure rate)
q: fraction of records not
discarded by a processor



$\alpha(\text{Branch}) = q \cdot \alpha(\text{Departure})$

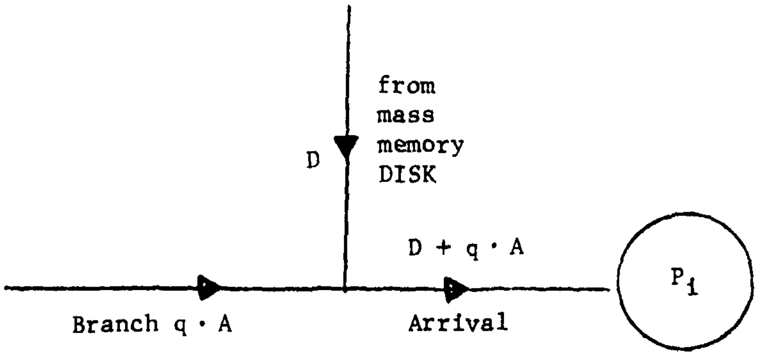
Figure A



δ = utilization factor of service facility

$\alpha(\text{Departure}) = \delta^2 \cdot \alpha(\text{service time}) + (1-\delta^2)\alpha(\text{Arrival})$

Figure B



$$\alpha(\text{Arrival}) = \left(\frac{E(\text{Arrival})}{E(\text{Disk})}\right)^2 \alpha(\text{Disk}) + \left(\frac{E(\text{Arrival})}{E(\text{Branch})}\right)^2 \alpha(\text{Branch})$$

Figure C

Sevick's Approximations for Estimating the
Second Moment of the Arrival Rate

So,

$$\alpha(\text{Arrival}) = q^2 \alpha(\text{Branch}) \ \ldots \qquad (3)$$

Solving equations 1, 2 and 3 simultaneously,

$$\alpha(\text{Arrival}) = \frac{q^3 \delta^2 \alpha(\text{service time})}{1-q^3(1-\delta^2)}$$

So,

$$\sigma^2(\text{Arrival}) = \frac{(1-q)^2 n^2 r^2}{p^2} \left(\frac{q^3 \delta^2 \alpha(\text{service time})}{1-q^3(1-\delta^2)} + 1 \right)$$

where, q the fraction of records not discarded by a processor is the only quantity to be evaluated. We conclude this section with a calculation of q.

As we know, a record is discarded by a processor if the AM indicates that it will not participate in the join (let us assume that a fraction a of records participate in the join) or it has been seen by all processors. Thus,

$$q = \frac{a\ E(\text{arrival rate from disk}) + \frac{n-2}{n-1} E(\text{arrival rate from previous processor})}{E(\text{arrival rate from disk}) + E(\text{Arrival rate from previous processor})}$$

$$= \frac{\frac{ap}{nr} + \frac{(n-2)qp}{(n-1)(1-q)nr}}{\frac{p}{nr} + \frac{qp}{(1-q)nr}}$$

$$= \frac{a}{1 + a - \frac{n-2}{n-1}}$$

Without the use of AM, a = 1, and, hence,

$$q = \frac{n-1}{n} \ .$$

APPENDIX 3: THE HEAVY TRAFFIC APPROXIMATION

We are interested in determining how large the B-memory should be in order to be 95% certain that it will not overflow (similar calculations can be made to determine the 99% limit, the 99.9% limit, or any other limit). Let us designate this B-memory size as $S_B 95$. $S_B 95$ may be calculated by determining the first and second moments of the queue length, and then using Chebychev's inequality [7]. The heavy traffic approximation to the G/G/1 queue tells us that the average queue length $\bar{S}_B$ is given by

$$\bar{S}_B = \frac{\lambda^2(\sigma^2_{\text{arrival}} + \sigma^2_{\text{service}})}{2(1-\delta)}$$

where,

$\delta = \lambda\bar{x}$

$\lambda =$ E(arrival rate)

$\bar{x} =$ E(service time)

$\sigma^2_{\text{arrival}}$ = second moment of interarrival time

and,

$\sigma^2_{\text{service}}$ = second moment of service time.

All the above quantities have been calculated in Appendix 2, and, hence, $\bar{S}_B$ the average queue length can be calculated.

The heavy traffic approximation gives the second moment of the queue length as $(\bar{S}_B)^2$. An application of Chebychev's inequality gives us

$$S_B 95 = \bar{S}_B(1 + \frac{1}{.05}) = 5.4721\ \bar{S}_B$$

APPENDIX 4: THE DIFFUSION APPROXIMATION

As in Appendix 3, we are interested in determining how large the B-memory should be in order to be 95% certain that it will not overflow. We will call this B-memory size Q_B95. In order to calculate Q_B95, we shall calculate the first and second moments of the queue length and then apply Chebychev's inequality. The Diffusion approximation [10] tells us that the probability of finding q elements in the queue is given by

$$P(q) = (1-\hat{\delta})\hat{\delta}^q$$

where,

$$\hat{\delta} = e^s$$

$$s = \frac{-2(1-\delta)}{\dfrac{\sigma^2_{service}}{\bar{x}^2} \times \dfrac{\sigma^2_{arrival}\ \delta}{(E(\text{interarrival time}))^2}}$$

$$\delta = \lambda\bar{x}$$

$$\lambda = E(\text{arrival rate})$$

$$\bar{x} = E(\text{service time})$$

$$\sigma^2_{service} = \text{second moment of service time}$$

and,

$$\sigma^2_{arrival} = \text{second moment of interarrival time}$$

A. Calculating the First Moment

$$E(q) = \sum_{q=o}^{\infty} q(1-\hat{\delta})\hat{\delta}^q$$

$$= (1-\hat{\delta})\frac{\hat{\delta}}{(1-\hat{\delta})^2}$$

$$= \frac{\hat{\delta}}{1-\hat{\delta}}$$

Thus,

$$\bar{Q}_B = \text{mean queue length} = \frac{\hat{\delta}}{1-\hat{\delta}}$$

B. Calculating the Second Moment

$$E(q^2) = \sum_{q=o}^{\infty} q^2(1-\hat{\delta})\hat{\delta}^q$$

$$= \frac{2\hat{\delta}}{(1-\hat{\delta})^2} - \frac{\hat{\delta}}{1-\hat{\delta}} \quad \text{(after much simplification)}$$

Thus,

$$E(q^2) = \bar{Q}_B \left(\frac{1+\hat{\delta}}{1-\hat{\delta}}\right)$$

Thus,

$$\sigma^2(q) = E(q^2) - (E(q))^2$$

$$= \text{second moment of queue length}$$

$$= \bar{Q}_B \left(\frac{1}{1-\hat{\delta}}\right)$$

C. Calculating Q_B95

Using Chebychev's inequality, we have

$$Q_B95 = \bar{Q}_B + \sqrt{\frac{\bar{Q}_B}{(1-\hat{\delta}) \cdot 05}}$$

ENI